jamk.fi

Synteettisen taulukkodatan generointi GAN-verkolla

Iikka-Matti Paakki

Opinnäytetyö
Tammikuu 2020
Tekniikan ala
Insinööri (AMK), Tieto- ja viestintätekniikka

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Sukunimi, Etunimi Paakki, Iikka-Matti	Opinnäytetyö, AMK	13.1.2020
	Sivumäärä 55 + 49	Julkaisun kieli Suomi
	-	Verkkojulkaisulupa myönnetty: x

Työn nimi

Synteettisen taulukkodatan generointi GAN-verkolla

Tutkinto-ohjelma

Tieto- ja viestintätekniikka

Työn ohjaaja(t)

Mika Rantonen, Sampo Kotikoski

Toimeksiantaja(t)

Tieto Finland

Tiivistelmä

Opinnäytetyön tavoitteena oli generoida uutta synteettistä taulukkomuotoista dataa alkuperäisen datan pohjalta. Generoidun synteettisen data tulisi olla tilastollisesti samanlainen kuin alkuperäinen.

Usein dataa ei voida käyttää yksityisyyden suojaamisen takia (esim. terveydenhuolto data). Generoimalla synteettistä dataa pyritään välttämään yksityisyyden suojan rikkomista. Uutta generoitua dataa voidaan käyttää mm. data-analytiikassa, koneoppimisessa ja sovelluskehityksessä ja sitä voidaan jakaa muille osapuolille ilman pelkoa yksityisyyden suojan rikkomisesta.

Opinnäytetyön teoriaosuudessa käydään läpi tekoälyä, koneoppimista ja syväoppimista. Lisäksi käydään läpi tarkemmin neuroverkon (syväoppimisen) rakennetta ja sen keskeiset toiminnalliset ominaisuudet. Lopuksi perehdytään GAN-verkon toiminnallisuuksiin sekä sen vahvuuksiin ja heikkouksiin.

Opinnäytetyön toteutusosiossa käydään läpi työn keskeiset käytännön vaiheet. Tarkemmin tarkastellaan erilaisen datan generoinnissa käytettävää GAN-verkkoa ja sen rakennetta. Lopuksi käydään läpi metodit, joiden avulla vertaillaan alkuperäistä sekä generoitua dataa keskenään.

GAN-verkon todettiin olevan toimiva ratkaisu homogeenisen numeerisen ja kategorisen datan generoinnissa. Generoitu data oli hyvin samankaltaista kuin alkuperäinen työssä käytettyjen vertailu metodien perusteella ja generoitu data voisi korvata alkuperäisen datan.

Avainsanat (asiasanat)

Koneoppiminen, Syväoppiminen, Neuroverkko, Generatiivinen kilpaileva verkko

Muut tiedot (Salassa pidettävät liitteet)

jamk.fi

Description

Author(s) Last name, First name Paakki, Iikka-Matti	Type of publication Bachelor's thesis	Date January 2020
		Language of publication: Finnish
	Number of pages 55 + 49	Permission for web publication: x

Title of publication

Synthetic tabular data generation with GAN

Degree programme

Information and Communication Technology

Supervisor(s)

Rantonen Mika, Sampo Kotikoski

Assigned by

Tieto Finland

Abstract

The aim of the study was to generate new synthetic tabular data from the original data. The generated synthetic data should be statically similar to the original data.

Often, data cannot be used for privacy reasons (e.g. healthcare data). The purpose of generating synthetic data is to avoid invasion of privacy. The newly generated data can be used for example in data analytics, machine learning, and application development, and it can also be freely distributed to others without fear of breach of privacy.

The theory part deals with artificial intelligence, machine learning and deep learning. In addition, the structure of the neural network (deep learning) and its main functional properties are examined in more detail. Finally, the functionalities of the GAN network and its strengths and weaknesses are examined.

The implementation part of the study covers the main practical steps of the thesis. The GAN network used for generating the different data and its structure is examined in more detail. Finally, the methods by which the original and generated data are compared are examined.

The GAN was found to be a viable solution for generating homogeneous numerical and categorical data. The generated data was very similar compared to the original data based on the comparison methods used in the thesis and could even replace the original data.

Keywords/tags (subjects)

Machine Learning, Deep Learning, Neural Network, Generative Adversarial Network

Miscellaneous (Confidential information)

Sisältö

Kuviot

1 Johdanto

1.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana toimi Tieto Finland. Tieto on ohjelmisto- ja palveluyritys, jolla on noin 15000 työntekijää 20 eri maassa, pääkonttori sijaitsee Espoossa. Tieto pyrkii datan ja avoimen lähdekoodin avulla innovoimaan yhteiskunnallisia muutoksia sekä tuottamaan taloudellista lisäarvoa sen asiakkaille. Tiedon asiakkaat tuottavat palveluita yrityksille sekä niiden asiakkaille eri puolilla maailmaa ja ovat pääasiassa suuria organisaatioita. Esimerkiksi Tieto ja Kanta-Hämeen sairaanhoitopiiri ovat yhteistyössä kehittämässä tekoälyn avulla parempaa ehkäisy- ja hoitosuunnitelmaa aivohalvauspotilaille, käyttäen potilaiden taustietoa paremmin ja tehokkaammin. Tieto valittiin maailman top-100 teknologiayritysten joukkoon vuonna 2018. (Meistä n.d.)

1.2 Toimeksianto

Nykypäivänä kerätään dataa hyvin paljon, eikä sitä pystytä käyttämään parhaalla mahdollisella tavalla yksityishenkilöiden yksityisyyden turvaamisen vuoksi, kuten terveyteen liittyvää dataa, tähän ongelmaan ratkaisuna on synteettinen data. Synteettistä dataa voidaan käyttää moniin käyttötarkoituksiin. Sitä voidaan jakaa toisille osapuolille sen anonymiteetin takia, sitä voidaan generoida määrittelemätön määrä ja sitä voidaan hyödyntää koneoppimisessa, data-analytiikassa, sovelluskehityksessä ja ehkä tärkeimpänä koulutusdatan lisäämisessä. Esimerkiksi kuvantunnistusongelmissa tarvitaan paljon dataa, joka johtaa taas parempiin lopputuloksiin.

Opinnäytetyön toivottu lopputulos muuttui työtä tehtäessä jonkin verran. Aluksi oli tarkoitus tutkia ja vertailla erilaisten generatiivisten neuroverkkojen mahdollisuuksia synteettisen datan luonnissa. Aihe todettiin olevan laaja ja haastava, myöskin hyvien testitulosten ansioista numeerisen datan syntetisoinnissa generatiivisella kilpailevalla verkolla (engl. Generative adversarial network, GAN), päädyttiin tutkimaan GAN-verkon mahdollisuuksia datan syntetisoinnissa. Työssä pyrittiin selvittämään voiko

GAN-verkkoa käyttää generoimaan kategorista ja numeerista taulukkodataa. Lisäksi työssä havainnoidaan GAN-verkolla generoitaessa ilmeneviä puutteita ja ongelmia.

Teoriaosuudessa käsitellään lyhyesti, mitä tekoäly on ja siihen lukeutuva koneoppiminen sekä koneoppimiseen kuuluva syväoppiminen. Erikseen käsitellään neuroverkon (syväoppimisen) perustoiminnallisuuksia ja ominaisuuksia. Lopuksi käsitellään generatiivisia kilpailevia neuroverkkoja, joihin työ keskittyy.

Opinnäytetyön toteutusosiossa käydään läpi työn keskeiset käytännön vaiheet. Tarkemmin tarkastellaan erilaisen datan generoinnissa käytettävää GAN-verkkoa ja sen rakennetta. Lopuksi käydään läpi metodit, joiden avulla vertaillaan alkuperäistä sekä generoitua dataa keskenään.

1.3 Tutkimusasetelma

Työn tavoitteena oli luoda synteettistä taulukkomuotoista dataa GAN-verkolla, joka muistuttaa tilastollisesti alkuperäistä dataa, mutta ei sisällä alkuperäisestä datasta ainuttakaan mittauspistettä, josta voitaisiin tunnistaa yksityishenkilö. Lisäksi oli tärkeää testata ja analysoida GAN-verkon lähestymistapoja ja rajoja taulukkomuotoisen datan syntetisoinnissa.

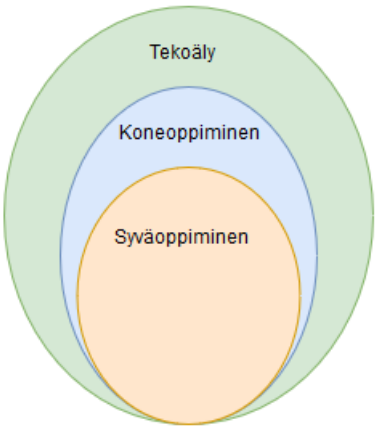
Tutkimusmenetelmäksi valittiin soveltava tutkimus. Työ jäsenneltiin seuraaviin kysymyksiin, joihin pyrittiin vastaamaan: Pystyykö GAN-verkolla generoimaan synteettistä numeerista ja kategorista dataa? Mitä rajoitteita/ongelmia esiintyy GAN-verkolla generoitaessa synteettistä dataa? Onko generoidulla synteettisellä datalla käytännön hyötyjä?

Työssä hyödynnettiin syväoppimisesta sekä GAN-verkoista löytyvää tietoa/tutkimuksia ja pyrittiin soveltamaan tätä aineistoa esitettyihin ongelmakysymyksiin käytännön sovelluksien kautta.

2 Tekoälystä syväoppimiseen

2.1 Tekoäly

Syväoppiminen on koneoppimisen menetelmä, ja koneoppiminen kuuluu tekoälyn menetelmiin (ks. kuvio 1). Termit voivat aiheuttaa hämmennystä keskenään. On siis hyvä käydä läpi, mitä nämä termit tarkoittavat ja minkälaisiin ongelmanratkaisuihin niitä käytetään, jotta voi ymmärtää paremmin, miksi kyseisiä menetelmiä käytettiin työssä ja minkälaisia tuloksia niistä olisi odotettavissa.



Kuvio 1. Syväoppiminen tekoälyssä

Tekoäly voidaan määritellä monella eri tapaa (ihmiset määrittelevät älykkyyden eri tavalla), ja se herättää paljon eriäviä mielipiteitä, eikä termi tekoäly kerro juurikaan mitään konkreettista siitä. Tekoälyllä pyritään simuloimaan ihmisen älykkyyttä käyttäen algoritmeja ja suorittamaan erilaisia toimintoja, kuten suorittamaan tehtäviä ilman jatkuvaa ohjausta samalla oppien menneisyyden kokemuksista ja parantamaan suorituskykyään jo opitusta. Tekoälyä käytetään yllättävän paljon nykymaailmassa, jopa niin paljon että ihmiset eivät ymmärrä käyttävänsä tekoälyä tai sen olemassa oloa. Esimerkiksi auto voi taskuparkkeerata itsensä autojen väliin, myöskin Googlen hakukone hyödyntää tekoälyä. Tekoäly voidaan luokitella neljään eri kategoriaan, joita on jo nykyaikana ja joita tulevaisuudessa mahdollisesti on (Massaron & Mueller 2018, luku 1):

Reaktiiviset koneet luottavat puhtaasti tietokoneen laskentatehoon tekemään paras mahdollinen ratkaisu sillä hetkellä, eivätkä ne omaa muistia menneisyyden tapahtumista ja päätöksistä. Eli ne eivät voi oppia tekemään päätöksiä aikaisemmista päätöksistä. Esimerkiksi vuonna 1997 reaktiivinen tietokone voitti maailmanmestari shakkipelaajan monen päivän taistelun jälkeen (Deep Blue n.d.).

Rajallisen muistin omaavat koneet voivat käyttää menneisyyden kokemuksia tekemään uusia päätöksiä, mutta pitävät tiedon vain rajallisen ajan hallussaan. Esimerkiksi itseohjautuva auto (tässä vaiheessa nykyaikainen tekoäly on).

Mielen teoria pyrkii ymmärtämään ympäristössä tapahtuvia tapahtumia, tekojen seuraamuksista ihmisen ajatuksia jne, toisin sanoen ihmisen mieltä. Tämä tekoälyn ala ei ole vielä kehitetty, vaikkakin sinne päin ollaan menossa.

Itsetietoiset koneet ovat kuin ihmiset. Ne ovat itsetietoisia, tekevät päätöksiä tunteiden perusteella jne. Tällainen tekoäly on mahdollista tällä hetkellä vain elokuvissa, eikä nykypäivänä olla lähelläkään itsetoisen koneen luomisessa.

2.2 Koneoppimien

Koneoppiminen on yksi monista tekoälyn osa-alueista. Koneoppimisella pyritään simuloimaan ihmisen oppimista siten, että ohjelmisto oppii suuresta määrästä dataa käyttäen algoritmeja parantaen suorituskykyään dataa lisättäessä. Se pystyy sopeutumaan epävarmoihin tilanteisiin ilman, että sitä ohjelmoidaan tekemään näin, ja lopuksi antaa vastauksen. Oppimisalgoritmi ei muutu missään vaiheessa, vaan pysyy samana. Kun dataa lisätään, algoritmin painotetut arvot ja vinoumat muuttuvat, ja näin algoritmi oppii. (Massaron & Mueller 2019, luku 2.)

Esimerkiksi: käyttäjä luo Netflixiin tunnuksen ja aloittaa katselemaan komediapainotteista sarjaa. Käyttäjän katsoma sarja lisätään sivuhistoriaan, josta algoritmi oppii käyttäjän pitävänä komediasarjoista. Sitten ohjelmisto vertailee niitä muiden samantyyppisten sarjojen pitävien käyttäjien sivuhistoriaan ja suosittelee uusia komediapainotteisia sarjoja, joista muut käyttäjät ovat pitäneet. (Yu 2019.)

Koneoppiminen käyttää siis algoritmeja oppiakseen tiedosta. Nämä koneoppimisen algoritmit voidaan jakaa karkeasti neljään pääjoukkoon: ohjattu oppiminen (engl. supervised learning), ohjaamaton oppiminen (engl. unsupervised learning), puoliohjattu oppiminen (engl. semi-supervised learning) ja vahvistusoppiminen (engl. reinforcement learning). (Massaron & Mueller 2019, luku 2.)

Ohjatussa oppimisessa tieto on merkitty (engl. label) luokkamuutujilla(ks. kuvio 2). Algoritmi koulutetaan tällä datalla, ja se tekee ennalta määritetyn ennusteen. Algoritmi oppii sitä mukaan, kun se tekee ennusteita ja saa niistä palautteita. Koulutusta jatketaan niin pitkään, että malli saavuttaa tietyn ennalta määritetyn tarkkuuden harjoitustietoon nähden. Ohjattua oppimista voidaan soveltaa luokittelu- ja regressio-ongelmiin. (Mt.)

Ohjaamattomassa oppimisessa syötetietoa ei ole merkitty luokkamuuttujilla (ks. kuvio 2) eikä haluttua lopputulosta tiedetä. Ohjaamatonta oppimista voidaan käyttää esimerkiksi mallintamaan tiedon luokittelua ja kaavojen etsimiseen. (Mt.)

Puoliohjatussa oppimisessa syötetieto on sekoitus merkittyjä ja merkittömiä luokkamuuttujia (ks. kuvio 2). Eli se on sekoitus ohjatun ja ohjaamattoman oppimisen menetelmiä. Etuja tässä on se, että suuria määriä dataa on hyvin raskasta ja aikaa kuluttavaa ”merkitä” kuuluvaksi johonkin luokkaan. Lisäksi merkitön data yleensä nostaa mallin tarkkuutta mallia kouluttaessa. Esimerkkikäyttökohteita puoliohjatulle oppimiselle ovat luokittelu ja ulottuvuuden vähentäminen. (Mt.)

Vahvistus oppiminen perustuu palkkiosysteemiin. Mallin suoriutuessa tehtävästä oikein se saa positiivisen palautteen ja näin vahvistaa oppimistaan. Huonosta suorituksesta malli saa negatiivisen palautteen ja osaa saa vahvistusta siitä, miten ei pitäisi toimia. (Mt.) Voidaan soveltaa esimerkiksi peleihin.

Ohjattu oppiminen			
ominaisuus 1	**ominaisuus 2**	**ominaisuus 3**	**luokkamuuttuja**
xxx	**xxx**	**xxx**	**yes**
xxx	**xxx**	**xxx**	**no**
xxx	**xxx**	**xxx**	**no**
xxx	**xxx**	**xxx**	**yes**
xxx	**xxx**	**xxx**	**no**
...	**...**	**...**	**...**
Ohjaamaton oppiminen			
ominaisuus 1	**ominaisuus 2**	**ominaisuus 3**	
xxx	**xxx**	**xxx**	
xxx	**xxx**	**xxx**	
xxx	**xxx**	**xxx**	
xxx	**xxx**	**xxx**	
xxx	**xxx**	**xxx**	
...	**...**	**....**	
Puoliohjattu oppiminen			
ominaisuus 1	**ominaisuus 2**	**ominaisuus 3**	**luokkamuuttuja**
xxx	**xxx**	**xxx**	**yes**
xxx	**xxx**	**xxx**	
xxx	**xxx**	**xxx**	**no**
xxx	**xxx**	**xxx**	
xxx	**xxx**	**xxx**	
...	**...**	**...**	

Kuvio 2. Erilaisten datakehysten soveltaminen koneoppimisen algoritmeille

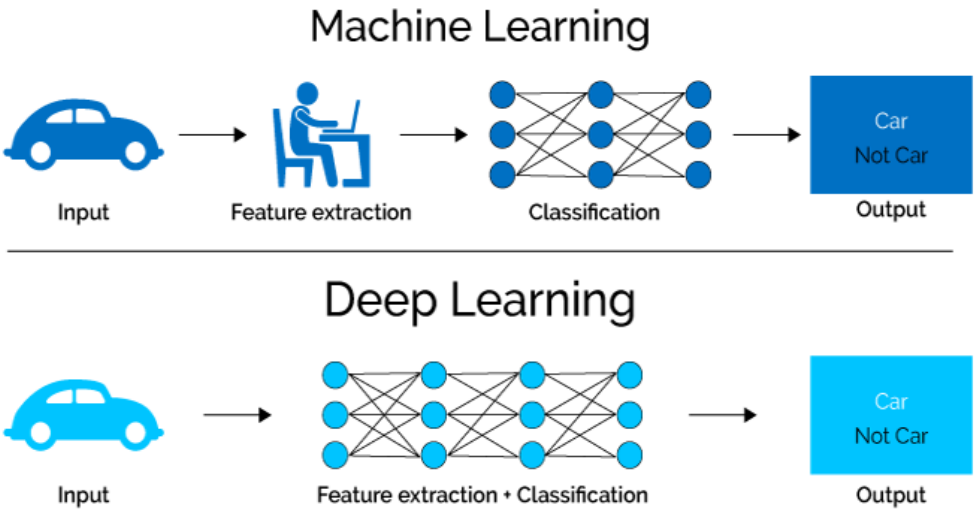
2.3 Syväoppiminen

Syväoppiminen kuuluu koneoppimisen osa-alueeseen. Koneoppimisessa ohjelma oppii datasta, niin myöskin syväoppimisen mallit. Kuitenkin näillä kahdella on paljon erilaisia ominaisuuksia ja niitä käytetään erilaisiin ongelmanratkaisuihin. Molemmat oppivat siis niille syötetystä datasta, mutta koneoppiminen käyttää monia erilaisia oppimismenetelmiä, kuten tilastollista analyysiä, kaavojen löytämistä datasta ja logiikkaa. Syväoppiminen sen sijaan käyttää vain yhtä oppimismenetelmää, jota kutsutaan neuroverkoksi. (Massaron & Mueller 2019, luku 1.)

Molemmissa oppimismenetelmissä on nk. hyperparametrejä, joiden avulla voidaan optimoida algoritmi oppimaan tarkemmin. Syväoppimisessa on myös lisänä nk. kerroksia, jotka ohjelmoija lisää manuaalisesti. Nämä kerrokset sisältävät yksinkertaisia neuroneja, jotka ovat yhteydessä seuraavan kerroksen neuroneihin, ja niiden yhteys

muodostaa neuroverkoston. Syväoppiminen saa nimensä neuroverkoissa sijaitsevista monista piilotetuista kerroksista, joihin neuronit on järjestetty. (Mt.)

Koneoppimisessa myös suoritetaan prosessi nimeltä ominaisuuksien luonti (engl. feature creation) ennen datan syöttämistä algoritmille (ks. kuvio 3). Esimerkiksi molemmille oppimismenetelmille annetaan tehtäväksi luokitella kissa- ja koirakuvia, jotka ovat samassa kokoelmassa. Koneoppimisen menetelmät eivät pysty kertomaan, mitkä kuvista ovat koiria ja mitkä kissoja ilman, että niille kerrottaisiin etukäteen, mitä ominaisuuksia niillä on. Syväoppimisen menetelmät eivät sen sijaan tarvitse ominaisuuksien luontia piilotettujen kerroksien ansioista, joissa algoritmit oppivat erottelemaan kuvista erilaisia ominaisuuksia, kuten koiran naama, kuono jne. Näiden ominaisuuksien takia syväoppiminen on parempi ratkaisu esimerkiksi kuvien tunnistuksessa, puheen tunnistuksessa ja kielten kääntämisessä. (Kapoor 2019.)



Kuvio 3. Koneoppiminen vs. syväoppiminen (https://www.deeplearning-academy.com/p/ai-wiki-machine-learning-vs-deep-learning)

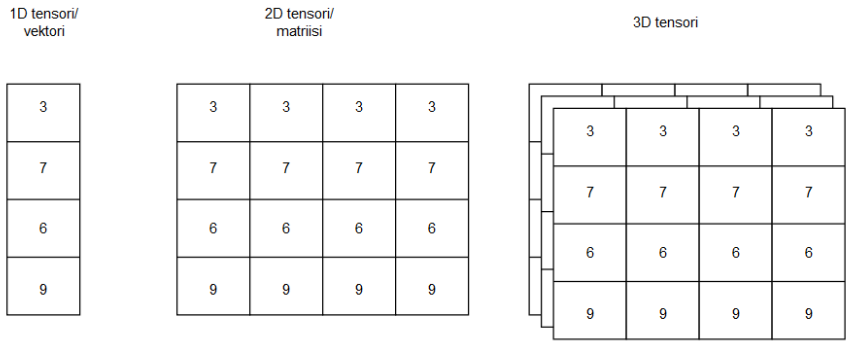
2.4 Syväoppiminen ja tietorakenteet

Tiedolla ja matematiikan avulla pyritään mallintamaan maailman monimutkaisuutta, esimerkiksi kissa voi olla numero 0 ja koira numero 1. Matematiikalla pyritään muuntamaan tietoa tietokoneelle ymmärrettävään muotoon, jotta löydettäisiin kaavoja ja rakenteita helpommin tiedosta. Esimerkiksi voit löytää tiedosta kaavan, että tietynikäiset ihmiset suosivat ennemmin koiria ja toiset taas kissoja. Tieto siis muokataan numeeriseksi matematiikan avulla tietokoneelle ymmärrettävään muotoon, varsinkin syväoppimisessa. Numeerinen tieto ei kerro tietokoneelle mitään, eikä se ymmärrä sitä ilman ihmisen tulkintaa. Ihminen muokkaa datan haluttuun muotoon esittäen oikean maailman abstraktioita. Sitten se syötetään syväoppimisen malliin. Malli suorittaa matemaatikkojen kehittämien todella monimutkaisten matemaattisten kaavojen avulla datan muokkausta ja antaa tuloksen, joka myöskin vaatii ihmisen tulkintaa. Tietokoneella ei siis ole ymmärrystä siihen syötetystä datasta, tiedon manipuloinnista ja lopputuloksesta. Syväoppimisella pyritään automatisoimaan tiedon tulkintaa ihmisten tapaan tietokoneen nopeudella. (Massaron & Mueller 2019, luku 5.)

Syväoppiminen käyttää matriisilaskentaa hallitakseen tiedon manipulointia. Matriisilaskennassa kerrotaan ja summataan järjestettyjä numerojoukkoja, joita on dataa käsitellessä skalaari, vektori ja matriisi. Skalaari on yksi data-arvo, kuten numero 1. Vektori on yksiulotteinen ryhmä, periaatteessa lista data-arvoja. Esimerkiksi vektori sisältää numeroarvoja ihmisten pituuksista 189, 150, 190 jne. Matriisi on kaksiulotteinen ryhmä data-arvoja, kuten taulukko, jossa on kolumneja ja rivejä (ulotteisuus kuvaa suuntaa). (Mt.)

Syväoppimisessa käytetään myös tensoritietorakennetta. Tensori voi olla skalaari, vektori, matriisi tai siinä voi olla loputon määrä ulottuvuuksia (ks. kuvio 4). Esimerkiksi kun kuva muutetaan numeeriseen muotoon syväoppimisen mallia varten, pikseleinä leveys ja korkeus, jotka voidaan esittää riveinä ja kolumneina, ja kolmanteen

ulottuvuuteen asetetaan kuvan värikoodi. Muunnoksen jälkeen tensori voidaan esittää kolmiulotteisena tensorina syväoppimisen malleihin. (Mt.)



Kuvio 4. Tietorakenteet

3 Keinotekoiset neuroverkot

3.1 Historia

Keinotekoiset neuroverkot (ANN) saavat inspiraationsa ihmisen aivojen neuroverkosta ja niiden toiminnallisuuksista. Tässä luvussa käydään läpi neuroverkkojen historiaa, neuroverkon kokonaisuutta sekä tarkastellaan hiukan, kuinka neuroverkon neuronit toimivat ja oppivat.

Vuonna 1957 psykologi ja tekoälyn tutkija Frank Rosenblatt kehitti perceptronin, joka on yksikertainen algoritmi ja se toimi vain yhdellä kerroksella. Perceptronia käytettiin binääriluokitteluun, esimerkiksi onko syöte koira vai ei, eli vastauksena saadaan joko 0 tai 1. Rosenblatt väitti perceptronin olevan alku uudelle tekoälylle, joka pystyisi kävelemään, puhumaan, kirjoittamaan, jopa lisääntymään ja olemaan itsetoinen. Kovista odotuksista huolimatta perceptron ei kuitenkaan saavuttanut sille asetettuja kovia odotuksiaan ja sillä huomattiin olevan rajoitteita (perceptron toimii nykyaina lähes kaikissa neuroverkoissa paranneltuna versiona neuronina). (Massaron & Mueller 2019, luku 7.)

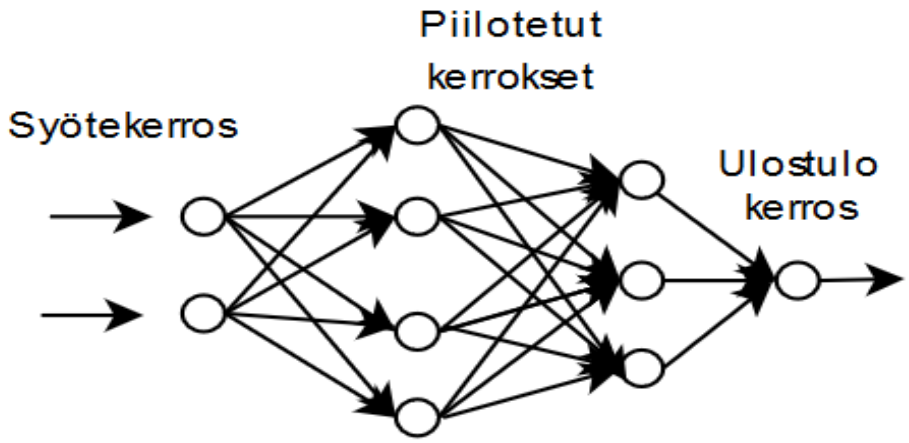
Kovan hypetyksen ansioista kiinnostus syväoppimista kohtaan laski, kunnes 1980-luvalla kehitettiin ensimmäinen neuroverkko monikerroksinen neuroverkko, joka käytti backpropagation algoritmia, jota käytetiin erilaisiin kuviontunnistusongelmiin. Kuitenkin neuroverkkojen koulutukseen meni huomattavan pitkä aika, johtuen tietokoneiden tehottomuudesta, eikä niillä ollut varsinaista käyttöä. Tietokoneiden laskentatehon ja uusien syväoppimismenetelmien kehittyessä syväoppiminen on herättänyt paljon mielenkiintoa tutkijoiden ja harrastelijoiden keskuudessa. Työhön keskittyvä GAN-verkko esiteltiin 2014 (ks. luku 5). (Marr 2019.)

3.2 Myötäkytkentäneuroverkko

Ihmisen aivot koostuvat neuronien yhteyksistä, kun taas neuroverkko rakennetaan neuronien yhteyksillä, jotka muodostavat verkoston. Jokainen neuroni linkittyy sisään- ja ulostulona toisiin neuroneihin. Neuronit lähettävät tietoa toisesta neuronista toiseen, vastaanottava neuroni prosessoi tiedon ja lähettää tiedon seuraavaan neuroniin. Neuronit on järjestetty hierarkkisesti kerroksiin, eivätkä ne ole yhteydessä saman kerroksen neuroneihin. Tietyt kolme kerrosta ovat aina mukana keinotekoisissa neuroverkoissa: syötekerros, piilotettu kerros (enemmän kuin yksi) ja ulostulokerros. (Massaron & Mueller 2019, luku 7.)

Kuviossa 5 nähdään syötekerros, jossa on kaksi solmua (esitetty ympyröinä). Se kertoo, että jokainen syöte on kaksiulotteinen (esim. ihmisen paino ja pituus). Seuraavat kaksi kerrosta ovat piilotettuja kerroksia, jossa on neljä ja kolme neuronia. Viimeisenä tulee ulostulokerros, jota myöskin voidaan kutsua ennustusluokaksi. Ulostulokerros antaa vastauksena yhden arvon tai arvoja sisältävän vektorin riippuen neuroverkosta ja sen käyttötarkoituksesta. Esimerkiksi voitaisiin käyttää neuroverkkoa luokittelemaan painon ja pituuden perusteella, onko ihminen ylipainoinen vai ei. Neuroverkko saa syötteenä painon ja pituuden. Syöte liikkuu neuroverkossa eteenpäin, kunnes se saavuttaa ulostulokerroksen ja antaa arvion, onko ihminen ylipainoinen vai ei. Kuviosta 5 voidaan huomata myös, että tieto kulkee vasemmalta oikealle nuolien

osoittamaan suuntaan. Tällaista neuroverkkoa kutsutaan myötäkytkentäneuroverkoksi (engl. feed forward neural network) neuroverkoksi. (Mt.)



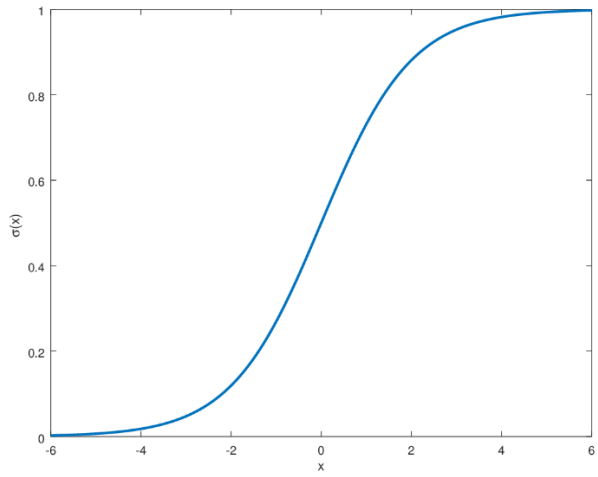
Kuvio 5. Monikerroksinen percetron verkko

3.3 Neuroni

Neuroni on neuroverkkojen keskeisin elementti. Nykyaikaiset neuronit saavat inspiraationsa aikaisemmin mainituista percetronista. Tiedemiehet huomasivat tutkiessaan neuroneja, että ne eivät aina lähetä signaalia eteenpäin. Kun neuroni vastaanottaa tarpeeksi stimulointia, se vapauttaa signaalin, muuten se pysyy hiljaisena. Esimerkiksi masennuslääkkeillä pyritään vahvistamaan aivojen neuronien välistä yhteyttä eli niiden aktiivisuutta. Keinotekoisten neuroverkkojen neuronit toimivat samalla tavalla: neuroni saa syötearvoja ja niiden vastaavia painoja (eng. weights), summaa ne ja esittää niiden painotetun summan aktivointifunktiolle. Painoarvo on jokin (pieni) numero ja tämä numero kuvastaa kahden neuronin välisen yhteyden voimakkuutta. Neuronilla on myöskin vakioarvo vinouma (engl. bias), joka auttaa mallia löytämään parhaan mahdollisen ratkaisun. Vinouma on valinnanvarainen eikä sitä ole pakko käyttää. Aktivointifunktio määrittelee milloin neuroni tai ryhmä neuroneja lähettää signaalin seuraavaan neuroniin, eli se määrittelee kynnysarvon, milloin signaali lähetetään. Aktivointifunktio muuntaa lopputuloksen arvon johonkin muotoon, yleensä epälineaariseksi. (Massaron & Mueller 2019, luku 7.)

Lineaarinen aktivointifunktio tuottaa tuloksena suoran viivan. Tulos voi olla – ääretömän ja + äärettömän välillä. Lineaarisella kaavalla ei voida ratkaista monimutkaisia funktionaalisia kartoituksia datasta sen yksinkertaisuuden takia. Aktivointifunktion puuttuessa neuroverkosta tai kun neuroverkko koostuu pelkistä lineaarisista aktivointifunktioista, neuroverkko on pelkkä lineaarisen regression malli, eikä sitä voida soveltaa esimerkiksi kuva- ja videodataan. (Massaron & Mueller 2019, luku 8.)

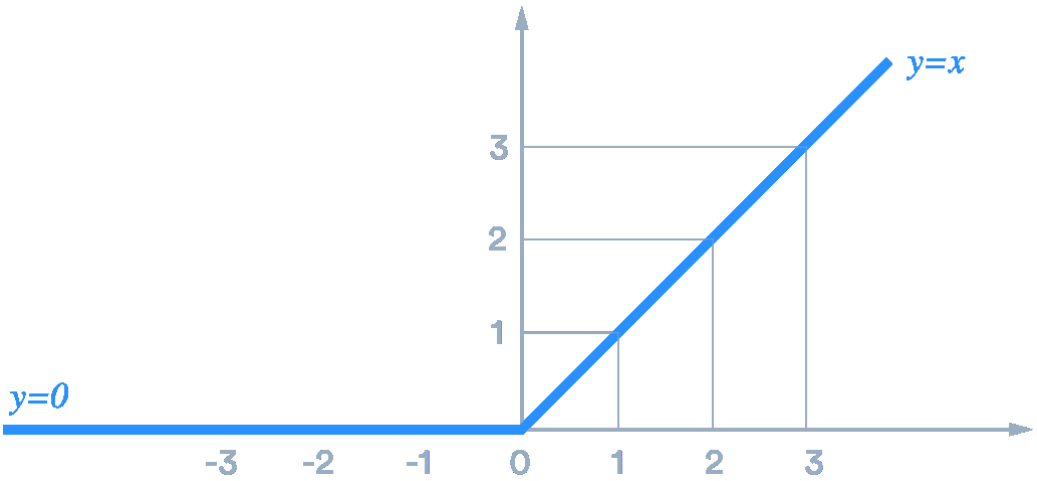
Sigmoid aktivointifunktio muuntaa tuloksen 0 ja 1 välillä ja tuottaa kuvion 6 mukaisen S muotoisen kuvion. Esimerkiksi jos muunnettu luku on 0.5, neuroni antaa ulostulona luvun 1 ja aktivoituu, 0,5 lukua pienemmät arvot eivät aktivoi neuronia, ja neuroni ei lähetä signaalia eteenpäin. (Mt.)



Kuvio 6. Sigmoid-aktivointifunktio (`A = 1 / 1 + e^-x`)

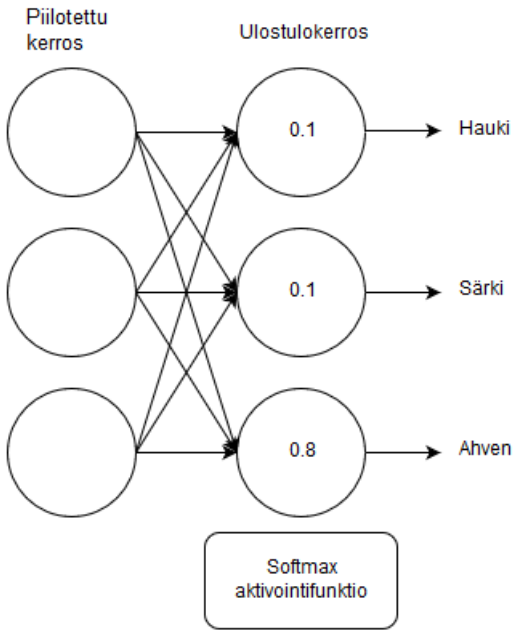
ReLu (engl. Rectified Linear Units) aktivointifunktio muuntaa ulostulon 0 ja äärettömän välille (ks. kuvio 7). ReLu:n etuna on, mitä enemmän luku on positiivinen, sitä

aktiivisempi se on. Kyseinen funktio voi myöskin aiheuttaa ongelmia jossakin vaiheessa, painot eivät enää vaikuta siihen halutulla tavalla, ja neuroni ei enää vatsaan syötteeseen. (Mt.)



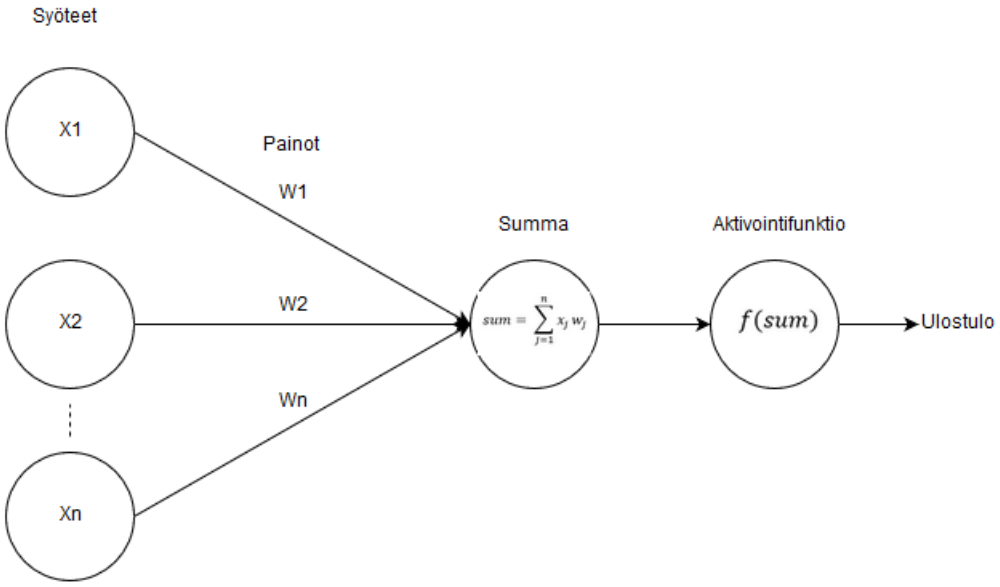
Kuvio 7. ReLu-aktivointifunktio (`A(x) = max(0, x)`)

Softmax-aktivointifunktio saa syötteenä vektorin arvoja ja muuntaa arvot 0 ja 1 välille eli todennäköisyysjakaumaan, kuten kuviossa 8. Arvojen summan on aina tasan 1. Softmaxia käytetään datan luokitteluun, jos luokkia on enemmän kuin kaksi, ja sitä käytetään usein ulostulokerroksen aktivointifunktiona. (Multi-Class Neural Networks: Softmax n.d.)



Kuvio 8. Softmax-aktivointifunktio

Käydään vielä läpi neuronin toimintaa seuraamalla kuvion 9 tapahtumia. Neuroni saa syötearvoja X1, X2, X3 ja niiden vastaavat painot W1, W2, W3. Tämän jälkeen lasketaan painotettu summa kaavan 1 mukaan. Kun painotettu summa on laskettu, se syötetään aktivointifunktiolle ja aktivointifunktio muuntaa painotetun summan valitun funktion mukaisesti kaavan 2 tapaan. Tämä prosessi tehdään neuroverkon kaikille neuroneille niin kauan, että päästään viimeiseen eli ulostulokerrokseen ja saadaan tulos. Kun useampi neuroni on käytössä, käytetään matriisilaskentaa, mutta käydään vain yksinkertainen yksittäisen neuronin esimerkki läpi.



Kuvio 9. Neuronin toiminta. Syötteet X, painot W, painotettua summa, aktivointifunktio ja ulostulo.

Esimerkkilasku käyttäen sigmoid-aktivointifunktiota ja kuvitteellisia lukuja:

$$sum = x_1 w_1 + x_2 w_2 + x_3 w_3 + b \quad (1)$$

$$0{,}4 * 0{,}8 + 0{,}3 * 0{,}7 + 0{,}7 * 0{,}9 + 0 = \ 1.0206$$

$$Sigmoid(sum) = (\frac{1}{1+ e^{-sum}}) \quad (2)$$

$$\frac{1}{1 + \ e^{-1.0206}} \sim 0{,}735$$

Kun käytetään sigmoid-aktivointifunktiota ja kynnyksenä on 0,5, niin neuroni lähettää signaalin eteenpäin.

3.4 Neuroverkon koulutus/oppiminen

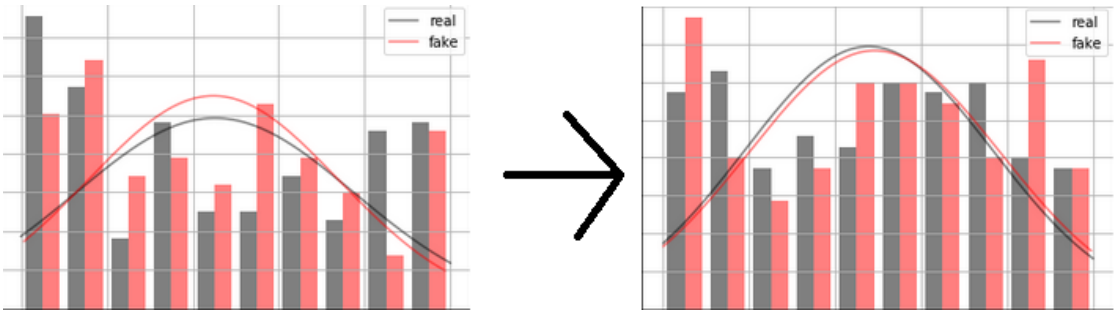
Neuroverkon oppiminen tapahtuu optimoimalla neuroverkon painoja. Optimoimalla painoja pyritään löytämään syötteen haluttu ulostulo, tätä prosessia kutsutaan neuroverkon oppimiseksi. Aikaisemmin mainittiin, että painot kuvastavat neuronien välistä yhteyden voimakkuutta, optimoimalla painoja saadaan halutut neuronit aktivoitua niin että saavutetaan haluttu lopputulos. Optimointialgoritmeja käytetään optimoimaan painoja ja ne määrittelevät miten painoja tulisi muuntaa, jotta päästäisiin lähemmäs haluttua lopputulosta, näistä yleisin on stokastinen kaltevuuden laskeminen (engl. stochastic gradient descent). Optimointi tapahtuu nk. häviöfunktion kautta, siten että painoja optimoidaan minimoimaan häviöfunktio mahdollisimman pieneksi tai suureksi riippuen käyttötarkoituksesta. Häviö kuvastaa ulostulon ja halutun ulostulon virhettä tai erotusta. Neuroverkoissa käytetään backpropagate nimistä algoritmia antamaan palautteen painoille, miten niitä tulisi optimoida, menemällä neuroverkossa taaksepäin. Eli neuroverkkoon syötetään alkuperäinen tieto ja neuroverkko antaa lopputuloksen, jonka jälkeen lasketaan häviöfunktio ja käytetään backpropagate algoritmia lähettämään takaisinpäin neuroverkkoon palaute, miten painoja tulisi säätää, riippuen optimointialgoritmista. Yhtä tällaista toimintakaavaa kutsunaan nimellä koulutussilmukka. Kun koulutussilmukoita suoritetaan monta kertaa, neuroverkko oppii sille ohjelmoidun tehtävän, tätä prosessia kutsutaan neuroverkon kouluttamiseksi. (Massaraon Mueller 2019, luku 7)

Neuroverkoissa jaetaan data erillisiksi opetus, validointi ja testijoukoiksi. Harjoitusdata syötetään neuroverkolle syötteenä uudestaan ja uudestaan, samalla se oppii ominaisuuksia kyseisestä datajoukosta. Samalla neuroverkko luokittelee validointidataa. Validointidataa käytetään tarkkailemaan, miten neuroverkko suoriutuu uuden datan luokittelussa. Yksi tärkeä syy validointidatan käyttöön on se, että sillä pyritään varmistamaan, ettei mallissa esiinny virhettä, jota kutsutaan ylisovittamiseksi. Ylisovittamisessa neuroverkko oppii luokittelemaan sen harjoitusdataa erittäin hyvin, mutta luokittelee sille uutta dataa erittäin huonosti. Siksi validointi dataa käytetään, jotta voitaisiin huomata ylisovittaminen ajoissa. Esimerkiksi neuroverkko luokittelee sen koulutusdatan 90% tarkkuudella, mutta validointidatan vain 40% tarkkuudella voidaan todeta neuroverkon olevan ylisovittunut. Ylisovittamista voidaan minimoida,

lisäämällä koulutusdataa, muuttamalla neuroverkon rakennetta, aktivointifunktiota, optimointifunktiota ja harjoitusiterointien määrää. Mallin koulutuksessa voidaan myös havaita virhe nimeltä alisovittaminen. Alisovittaminen on helppo huomata ja se voidaan todeta, kun neuroverkko luokittelee hyvin huonosti sen harjoitusdataa. Alisovittamista voidaan vähentää lisäämällä kerroksia neuroverkkoon, lisäämällä neuronien määrää kerroksissa ja muuttamalla neuroverkon kerroksia. Lopuksi, kun neuroverkko on koulutettu, käytetään sitä ennustamaan testidataa, jota se ei ole aikaisemmin nähnyt. Tämän prosessin avulla pyritään varmistamaan, että neuroverkko toimii halutulla tavalla uuden datan luokittelussa. (Train, Test, & Validation Sets explained 2017; Underfitting in a Neural Network explained 2017)

4 Generatiiviset mallit

Generatiiviset mallit ovat keränneet suurta suosiota nykyaikana niiden moninaisista mahdollisuuksista generoida uutta dataa, jota ei olla vielä nähty. Niitä käytetään jäljittelemään alkuperäisen tiedon todennäköisyysjakaumaa ja ideaali tulos olisi, ettei generoitua dataa voitaisi erottaa alkuperäisestä tai generoitu data olisi alkuperäistä dataa parempi. Koska malli oppii sen syötedatasta jakauman, voidaan sillä tuottaa uusia lukuja, jotka ovat alkuperäisen datan jakauman alueella. Generatiivisilla malleilla pyritään maksimoimaan alkuperäisen datan todennäköisyys generoidulle datalle, siten että pyritään löytämään minimaalisin etäisyys oikean- ja generoidun datan jakauman välillä kuvion mukaisella tavalla (ks. kuvio 10).



Kuvio 10. Jakauman etäisyyden minimointi

Generatiiviset mallit voidaan jakaa kahteen luokkaan: epäsuora koulutus ja suora koulutus. Suoran koulutuksen mallissa malli saa syötteenä alkuperäisen datan jakauman. Esimerkiksi variaatio autoenkooderit (engl. variational autoencoder) ovat tällaisia, ja niitä voidaan käyttää esimerkiksi poistamaan ”kohinaa” alkuperäisestä kuvasta. Epäsuora koulutusmetodin malli ei saa syötteenä alkuperäisen datan jakaumaa, vaan yleensä jostakin sattumanvaraisesta jakaumasta (latentista tilasta). Nämä mallit kouluttavat itseään kilpailevan prosessin avulla. Tällainen malli on generatiivinen kilpaileva verkko (GAN-verkko) (Hong, Hwang, Yoo & Yoon 2019, 1.)

5 Generatiiviset kilpailevat neuroverkot

5.1 Yleistä

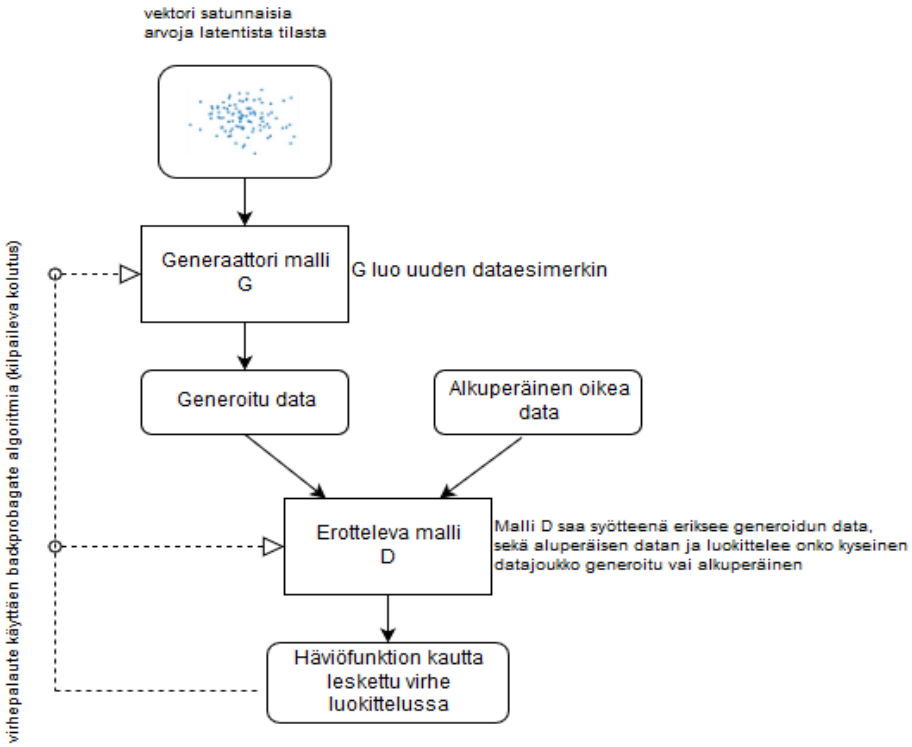
Ian Goodfellow ja muut tutkijat esittelivät 2014 uuden koneoppimismenetelmän, joka käyttää syväoppimisen tekniikoita. Uuden menetelmän nimi on generatiivinen kilpaileva verkko (GAN). Se arvioi kahta neuroverkkoa kilpailevan prosessin avulla. GAN on herättänyt paljon mielenkiintoa, koska sen avulla voidaan jäljitellä minkä tahansa datan jakaumaa, riippumatta datan ulottuvuuksista. Facebookin tekoälyn tutkimusjohtajan Yann LeCun mukaan GAN on kiinnostavin aihe kymmeneen vuoteen koneoppimisen saralla. Esimerkiksi GAN:ia käytettiin generoimaan uusi taideteos, joka myytiin New Yorkin huutokaupassa 432,500 dollarilla. (Nicholson n.d.)

GAN koostuu kahdesta erillisestä neuroverkosta: generatiivisestä (engl. generative) mallista G ja erottelevasta/luokittelevasta (engl. discriminative) mallista D, jotka kilpailevat keskenään. Malli G on vastuussa uudesta generoidusta datasta (”väärennetystä”), joka jäljittelee alkuperäisen datan jakaumaa. Malli D sen sijaan yrittää erottaa, onko sille syötetty data alkuperäistä vai väärennettyä. Molempia malleja koulutetaan mallin D avulla. Mallia G koulutetaan maksimoimalla todennäköisyys sille, että malli D tekee virheen, ja mallia D koulutetaan siten, että se pystyy luokittelemaan mahdollisimman hyvin, onko kyseessä aito datajoukko vai generoitu datajoukko, eli pyritään minimoimaan luokitteluvirhe. Malli G ei koskaan näe alkuperäistä dataa, ja

se sopeutuu sen mukaan, kuinka hyvin malli D suoriutuu. (Bengio, Courville, Goodfellow, Mirza, Ozair, Pouget-Abadie, Warde-Farley & Xu 2014, 1)

5.2 GAN:in toiminta

Tarkastellaan GAN:in toimintaa hiukan tarkemmin kuvion 11. avulla. Malli G saa syötteenä satunnaisen vektorin sisältäen latentteja arvoja, minkä jälkeen generoidaan uusi dataesimerkki. Tämän jälkeen alkuperäisestä datasta otetaan näyte, jonka koko vastaa generoitua dataa. Alkuperäistä ja generoitua dataa ei vertailla missään vaiheessa keskenään, vaan ne syötetään erikseen mallille D, joka arvio niiden aitouden. Yleensä malli D:n viimeisenä aktivointifunktiona käytetään sigmoid funktiota, joka muuntaa lopputuloksen 0 ja 1 välille. Arvo 1 mallissa D implikoi, että data on alkuperäinen, ja taas arvo 0 implikoi, että data on generoitua. Eli jos arvo < 0,5, se on generoitua dataa ja jos arvo > 0,5 se on alkuperäinen data. Ideaalissa lopputuloksessa malli D antaa arvon 0,5 ja silloin se ei enää erota alkuperäistä ja generoitua dataa toisistaan niiden samanlaisen jakauman ansioista. Aluksi malli D pystyy helposti erottamaan generoidun ja alkuperäisen datan toisistaan (kilpailevan koulutuksen ansioista datan luokittelusta tulee vaikeampaa). Luokittelun jälkeen lasketaan luokitteluvirhe, joka lähetetään takaisin malleille käyttäen backpropagate algoritmia.



Kuvio 11. Generatiivinen kilpaileva neuroverkko (vanilja versio)

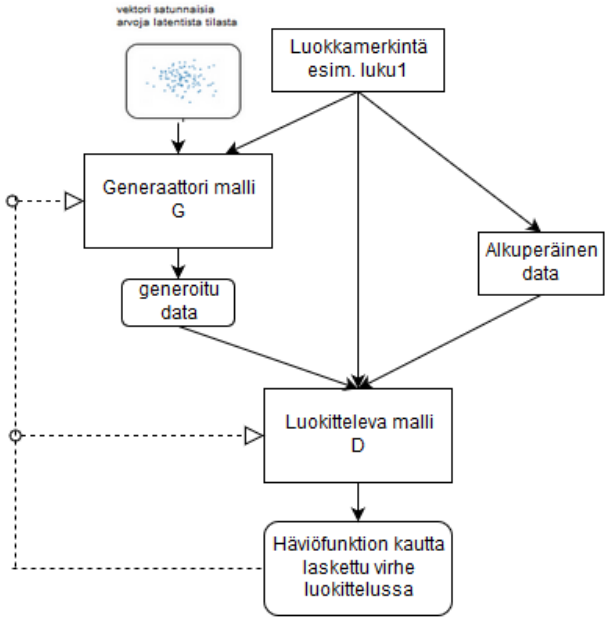
Luokitteluvirhettä käytetään parantamaan molempien mallien suoritusta. Kuten jo todettu, mallia D päivitetään siten, että se tunnistaa paremmin onko sille syötetty data alkuperäinen vai generoitu ja mallia G päivitetään sen mukaan, miten hyvin tai huonosti generoitu data huijaa mallia D. Tämä kuvailtu prosessi on vain yksi koulutussilmukka, ja kun prosessia toistetaan tuhansia kertoja, molemmat mallit saavat ”kokemusta” tehtävistään ja generoidusta datasta tulee hyvin samankaltainen kuin alkuperäisestä, jopa niin samankaltainen että niitä on mahdoton erottaa toisistaan pintapuolisesti tarkasteltuna. GAN:issa on siis kaksi kilpailevaa neuroverkkoa, toinen generoiva malli ja toinen perinteinen luokitteleva malli, ja niiden tarkoitus on voittaa toinen toisensa ja tästä nimi generatiivinen kilpaileva neuroverkko. (Brownlee 2019; Gan n.d.)

5.3 GAN:in hyvät ja huonot puolet

Yksi GAN:ien etuna on niiden kyky mallintaa moniulotteisen ja mutkikkaan datan jakaumaa. GAN:it ovat saavuttaneet suuren suosion niiden menestyksen ansioista kuvadatan erilaisissa ongelmanratkaisuissa. Kuten todettua, väärennöksistä tulee niin hyviä, että ihmiset eivät pysty toteamaan niitä väärennetyiksi pintapuoleisesti tarkasteltuna. Kun generoitu data on lähes samanlaista, sitä voidaan käyttää tietyissä ongelmanratkaisuissa kouluttamaan malleja, joissa ei ole tarvittavaa määrää dataa mallien koulutukseen. Näin saadaan malleista vieläkin tarkempia. Esimerkiksi GAN:in generoimaa dataa terveysdataa voidaan käyttää kouluttamaan uusia malleja ennustamaan mahdollisia sairauksia. (Goodfellow 2016, 3.)

Aikaisemmin kuvailtu GAN on nk. ”vanilja”(alkuperäinen) versio GAN:ista, se käyttää yksinkertaista monikerroksista percetron verkkoarkkitehtuuria generaattorissaan ja luokittelijassaan. Perinteisellä GAN:illa voidaan tuottaa mahdollisia uusia kuvia hyvin yksinkertaisista kuvista, joissa pikselimäärä on pieni. Pikselimäärän kasvaessa vaaditaan GAN:ilta enemmän vaihtelua toiminnallisuuksiin, kuten erilaisia verkon arkkitehtuuria ja häviöfunktioita. Toinen hyvä puoli on, että GAN:eista on julkaistu todella paljon tutkimuspapereita, joissa ehdotetaan uusia muokattuja GAN versioita, joiden ansioista generoidun datan laatu on merkittävästi parantunut, sekä GAN:in käyttökohteet ovat laajentuneet. (Khoshgoftaar & Shorten 2019, 22.)

Yksi yleisemmin käytetyistä GAN versioista on ”ehdollinen” generatiivinen kilpaileva neuroverkko (engl. conditional generative adversarial network, CGAN). Perinteisellä GAN:illa voidaan generoida mahdollisia uusia kuvia alkuperäisestä datasta, mutta ei voida kontrolloida sitä minkä tyyppistä dataa sillä generoidaan. CGAN:illa voidaan generoida johonkin luokaan kuuluva kuva ehdon avulla. Esimerkiksi johonkin luokkaan kuuluu kissakuvia ja ehdon avulla voidaan kohdentaa generaattori generoimaan uusia kissakuvia. Kuten kuviossa 12 huomataan luokkamuuttuja, jonka avulla voidaan generoida halutun luokan dataa. Muitakin ehdollisia GAN:eja on kuten SGAN (puoliohjattu GAN), infoGAN aj AC-GAN. (Mts. 25.)



Kuvio 12. CGAN

Kolmas hyvä puoli GAN:eissa on se, että niitä voidaan kouluttaa puuttuvalla datalla ja niitä voidaan käyttää puuttuvan datan ennustamiseen. Erityisesti huomiota on herättänyt aikaisemmin mainittu puoliohjattu GAN. Kuten luvussa 2.1 mainittu puoliohjattu oppiminen sisältää merkittyjä ja merkittömiä datapisteitä. Nykyaikana hyvin moni algoritmi luottaa siihen, että data on merkittyä, mutta puoliohjatun oppimisen etuna on vähäinen luokkamuuttujien määrä. Puoliohjattu GAN käyttää luokittelijaa (mallia D) ennustamaan uusia mahdollisia luokkamuuttujia ja niihin kuuluvaan dataa. (Goodfellow 2016, 3.)

Neljäs hyvä puoli on se, että GAN:eja voidaan muokata käsittelemään erilaista dataa samanaikaisesti, eli voidaan luoda molemmille malleille useampi syöte- tai ulostulokerros. Esimerkiksi generaattorin ulostulo voidaan muokata tuottamaan pituus ja ikä. Tällöin tarvitaan kaksi ulostulokerrosta generaattorille, ja jos jatketaan perinteisen GAN:in tapaan syöttämällä nämä kaksi ulostuloa syötteenä luokittelevalle mallille, niin luokittelevaan malliin tulisi muokata kaksi syötekerrosta. Muitakin hyviä puolia GAN:eista löytyy, mutta juuri käsitellyt ovat keskeisimpiä tämän tutkimuksen kannalta. (Mts. 4.)

Suurin negatiivinen puoli GAN:issa on se, että se on uusi idea ja tämän takia siinä on vielä tärkeitä ongelmia ratkaistavana. Kuten aiemmin mainittu GAN:issa ideaalitulos olisi se, että luokitteleva malli antaisi tuloksen 0.5, eli se ei enää tiedä onko kyseessä alkuperäinen tai väärennetty data kyseessä, toisin sanoen päätavoite on löytää tasapainotila. GAN:eja on vaikea kouluttaa, koska koulutus tapahtuu toisen mallin kustannuksella, eikä niin sanottua mallien ”lähentymistä” (engl. converge) tasapainotilaan aina saavuteta. Ehkä yleisin ongelma GAN:eja kouluttaessa on, kun generaattori generoi niin huonoja datajoukkoja, että erottelija huomaa aina kyseessä olevan väärennetty datajoukko. Toinen yleinen häiriö koulutuksessa on, kun generaattorimalli generoi sen saaman syötteen vain yhteen ulostulopisteeseen tai hyvin lähelle yhtä pistettä, tätä virhettä kutsutaan tilan romahtamiseksi (engl. mode collapse). Näitä ongelmia voidaan diagnosoida tarkastelemalla mallin koulutuksen yhteydessä saatuja virhepalautteita joko tulostamalla tai mallintamalla ne kuvaajaan. (Brownlee 2019b; Goodfellow 2016, 34.)

Yksi yleinen ongelmakysymys on, kuinka arvioida generatiivista mallia laadukkaasti. Tähän ei ole vielä löydetty toimivaa ja yleistä käytäntöä, vaikka monia tapoja onkin, ne eivät toimi täydellisesti ja sisältävät erinäisiä ongelmia. Yksi tutkimuksen kannalta tärkeä heikkous on se, ettei GAN:in generaattoreilla voida generoida kuin jatkuvia lukuja. Esimerkiksi työn kannalta tärkeää olisi pystyä generoimaan diskreettejä lukuja, koska data sisältää luokkamuuttujia (esim. maat: suomi, ruotsi jne.), toki tämä ongelma tässä käyttötarkoituksessa voidaan kiertää pyöristämällä luvut haluttuun lähimpään muotoon, kun uutta dataa on generoitu. (Goodfellow 2016, 42.)

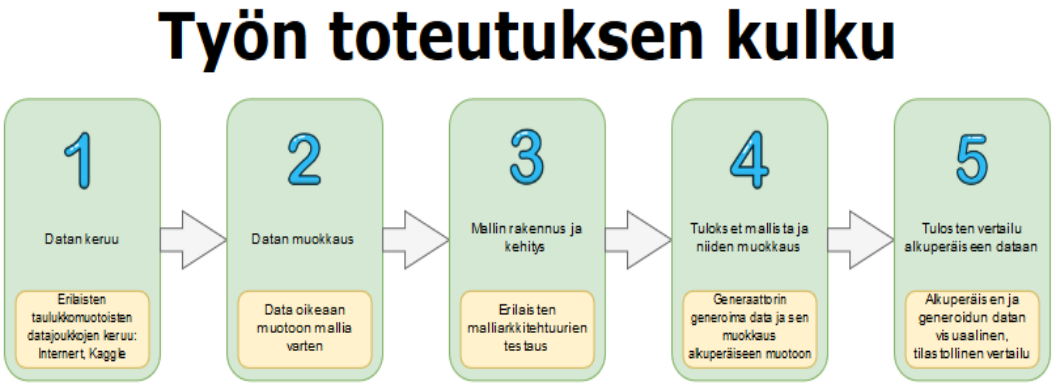
6 Toteutus

6.1 Toteutuksen kulku

Python valittiin ohjelmointikieleksi sen yksinkertaisuuden, suuren kehittäjäyhteisön ja laajan datatieteiden kirjaston määrän perusteella. Ohjelmointiympäristönä käytettiin Google Colaboratorya. Google Colaboratory toimii pilvessä ja se tarjoaa ilmaista

laskutehoa erilaisille datatieteiden sovelluksille ja siinä pystytään suorittamaan Python skriptejä nk. soluissa. Soluissa pystytään erittelemään työn erilaiset vaiheet ja niihin pystyy lisäämään tekstiosioita, jotka helpottavat työn vaiheiden seuraamista verraten perinteisiin ohjelmointitekstieditoreihin.

Työ sisälsi useita eri vaiheita ja ne jäsenneltiin viiteen eri vaiheeseen kuvion 13 kuvaamalla tavalla. Lisäksi kuvion 13 vaihe kolme jaoteltiin eri vaiheisiin: numeerisen datan generointi, kategorisen datan generointi ja numeerisen sekä kategorisen datan generointi. Näin saadaan tutkittua erikseen, miten GAN-verkko suoriutuu erilaisen datan generoinnissa ja voidaan tutkia niissä ilmeneviä ongelmia.



Kuvio 13. Työn toteutuksen kulku

6.2 Datakehyksen valinta ja sen muokkaus

Datakehys, josta GAN-verkko generoi uutta dataa, valittiin internetistä löydetty datakehys, johon lisättiin manuaalisesti uudet kolumnit. Alkuperäisessä datakehyksessä oli kategorinen kolumni, jossa muuttujat ”Male” ja ”Female”, lisäksi kaksi numeerista kolumnia, joissa paino ja pituus. Datakehykseen lisättiin kolumni, jossa ”potilaiden tunnukset” generoimalla satunnaisia kokonaislukuja. Lisäksi datakehykseen lisättiin painoindeksi kolumni, laskemalla käyttäen paino ja pituus kolumneja (ks. kuvio 14).

```
df.Height = (df.Height * 2.54) / 100
```

```
df.Weight = df.Weight * 0.45359237
```

```
df["bmi"] = np.round(df.Weight/ (df.Height * df.Height), 2)
```

Kuvio 14. Painoindeksin laskeminen

Datakehykseen lisättiin myös yksi kategorinen kolumni painoindeksiluokan perusteella. Kuviossa 15 määritellään raja-arvot mihin luokkaan tietyn painoindeksin omaava ”potilas” kuuluu. Datakehyksen muokkauksessa käytettiin Numpy ja Pandas kirjastoja.

```
bmilist = []
for bmi in df.bmi:
    if (bmi < 18.5):
        bmilist.append("underweight")

    elif ( bmi >= 18.5 and bmi < 25):
        bmilist.append("healthy")

    elif ( bmi >= 25 and bmi < 30):
        bmilist.append("overweight")

    elif ( bmi >=30):
        bmilist.append("severely overweight")

df["bmi_class"] = bmilist
```

Kuvio 15. Luokittelu painoindeksin perusteella

Datakehyksen valinnan ja muokkailun jälkeen data muokattiin GAN-verkolle oikeaan muotoon. Yleensä jatkuvat luvut skaalataan joko [-1, 1] tai [0, 1] lukujen välille. Jatkuvat luvut skaalattiin käyttäen Python sklearn MinMaxScaler luokkaa (ks. kuvio 16).

```
mms = MinMaxScaler()
numerical_data_rescaled = mms.fit_transform(numerical_data)
numerical_data_rescaled
```

```
array([[0.05449635, 0.32883004, 0.35449659, 0.52878888],
       [0.2713214 , 0.34286465, 0.46147653, 0.73395103],
       [0.56639992, 0.42129613, 0.41217124, 0.53408339],
       ...,
       [0.82789837, 0.60067446, 0.67514199, 0.79285242],
       [0.33531059, 0.65310969, 0.63883337, 0.66710788],
       [0.43494048, 0.39514045, 0.39973922, 0.54070152]])
```

Kuvio 16. Numeeristen muuttujien skaalaaminen

Kategoriset luvut muunnetaan käyttäen one-hot encode metodia (ks. kuvio 17). Näiden muokkausten avulla GAN-verkosta saadaan vakaampi ja paremmin suoriutuva. Muokattu data syötetään kaksiulotteisena Numpy taulukkona GAN-verkolle.

```
2  gender_ohe = pd.get_dummies(df.Gender)
3  gender_ohe
```

	Female	Male
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1
...	...	...
9995	1	0
9996	1	0
9997	1	0
9998	1	0
9999	1	0

10000 rows × 2 columns

Kuvio 17. One-hot encoded muuttujat

6.3 GAN-verkon rakenne ja koulutus

GAN-verkon luonnissa käytettiin Python kirjastoa nimeltä Keras. Keras on luotu neuroverkkojen ohjelmistorajapinnaksi, joka toimii Google kehittämän Tensorflowin päällä. Tensorflow on yksi suosituimmista koneoppimisen kirjaistoista, mutta sen on suhteellisen haastava oppia ja Keras tarjoaa tähän yksinkertaisemman ohjelmointiratkaisun. Keras käyttää siis Tensorflowin ominaisuuksia yksinkertaisemmalla koodilla. Keraksen avulla luotiin funktiot: generaattori, joka vastaa synteettisen datan generoinnista, erottelija/luokittelija vastaa sille syötetyn datan luokittelusta, generaattorin ja erottelijan yhdistäminen GAN-verkoksi, sekä GAN-verkon koulutus. GAN-verkon koulutuksen päätyttyä piirrettiin kuvaajaan koulutuksesta saadut virhepalautteet käyttäen Python matplotlib kirjastoa. Virhepalautteiden visualisoinnilla pyritään tarkastelemaan mahdollisia virheitä GAN-verkon koulutuksessa.

6.3.1 GAN-verkolla numeeristen muuttujien generointi

Yksinkertainen ongelmaesimerkki numeerisilla muuttujilla käyttäen *Keras sequential* mallia. *Sequential* malli on yksinkertainen neuroverkko lineaarisesti pinottuja kerroksia. Aluksi määritellään kuvion 18 (rivi 1) mukaisella tavalla generaattorifunktio ja sille syötettävät parametrit. Sitten alustetaan muuttujaan *sequential* malli, jonka jälkeen lisätään malliin syötekerros. Syötekerroksen suuruus riippuu funktion parametrin *laten_dim* suuruudesta (rivi 3). Syötekerroksen jälkeen lisätään syvät kerrokset ja niissä käytettävien neuronien määrä, sekä aktivointifunktiot.

```
1  def build_generator(n_columns, latent_dim):
2      model = Sequential()
3      model.add(Dense(32, kernel_initializer = "he_uniform", input_dim=latent_dim))
4      model.add(LeakyReLU(0.2))
5      model.add(BatchNormalization(momentum=0.8))
6      model.add(Dense(64,  kernel_initializer = "he_uniform"))
7      model.add(LeakyReLU(0.2))
8      model.add(BatchNormalization(momentum=0.8))
9      model.add(Dense(128,  kernel_initializer = "he_uniform"))
10     model.add(LeakyReLU(0.2))
11     model.add(BatchNormalization(momentum=0.8))
12     model.add(Dense(n_columns, activation = "sigmoid"))
13     return model
14
```

Kuvio 18. Numeeristen muuttujien generointi

Lopuksi lisätään ulostulokerros, joka sisältää aktivointifunktion ja funktion parametrin *n_columns* määrittelemän määrän neuroneja. Viimeisenä aktivointifunktiona käytetään *sigmoid* funktiota, koska halutaan generoida lukuja väliltä 0-1 (alkuperäinen data on skaalattu välille 0-1). Parametri *n_columns* vastaa alkuperäisen datakehyksen kolumnien määrää. Näin saadaan generoitua sama määrä kolumneja kuin alkuperäisessä. Generaattorifunktio palauttaa generaattorimallin ja se tallennetaan muuttujaan generaattorifunktiokutsun kautta myöhempää käyttöä varten.

Seuraavaksi määritellään luokittelijafunktio ja sille syötettävä parametri kuvion 19 mukaisella tavalla. Alustetaan muuttujaan *sequential malli*, jonka jälkeen lisätään malliin syötekerros. Syötekerroksen suuruus riippuu funktion parametrin *n_inputs* suuruudesta. Parametri *inputs_n* vastaa alkuperäisen sekä generoidun datan kolumninen määrää. Syötekerroksen jälkeen lisätään syvät kerrokset ja niissä käytettävien neuronien määrä sekä aktivointifunktiot.

```
def build_discriminator(inputs_n):
    model = Sequential()
    model.add(Dense(256,  kernel_initializer="he_uniform", input_dim=n_inputs))
    model.add(LeakyReLU(0.2))
    model.add(Dense(128,  kernel_initializer="he_uniform"))
    model.add(LeakyReLU(0.2))
    model.add(Dense(64,  kernel_initializer="he_uniform"))
    model.add(LeakyReLU(0.2))
    model.add(Dense(32,  kernel_initializer="he_uniform"))
    model.add(LeakyReLU(0.2))
    model.add(Dense(1, activation="sigmoid"))
    model.compile(loss="binary_crossentropy", optimizer=optimizer, metrics=["accuracy"])
    return model
```

Kuvio 19. Numeerinen luokittelija

Syvien kerroksien jälkeen lisätään malliin ulostulokerros, jossa on vain yksi neuroni ja se käyttää *sigmoid* aktivointifunktiota antamaan vastauksen, onko sille syötetty data generoitua vai alkuperäistä (ks. luku 5.2). Lopuksi määritellään luokittelijan virhefunktio ja optimointialgoritmi (ks. luku 3.4). Luokittelijafunktio palauttaa luokittelijamallin ja se tallennetaan muuttujaan luokittelijafunktiokutsun kautta myöhempää käyttöä varten.

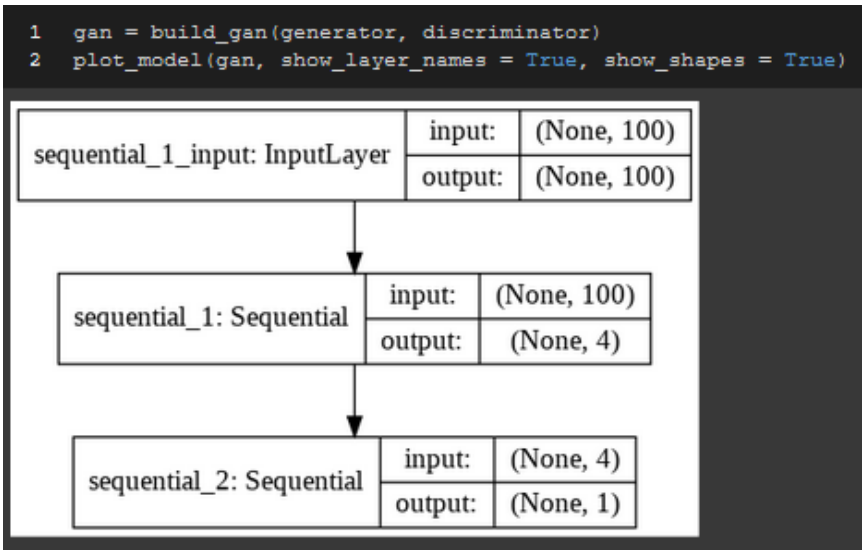
Kun generaattori ja luokittelija ovat valmiita ne voidaan ”liittää” yhdeksi kokonaisuudeksi eli GAN-verkoksi kuvion 20 mukaisella tavalla. GAN-funktio saa parametreiksi muuttujiin tallennetut generaattori ja luokittelija mallit.

```
def define_gan(generator, discriminator):
    # make weights in the discriminator not trainable
    discriminator.trainable = False
    # connect them
    model = Sequential()
    # add generator
    model.add(generator)
    # add the discriminator
    model.add(discriminator)
    # compile model
    model.compile(loss="binary_crossentropy", optimizer=optimizer)
    return model
```

Kuvio 20. Generaattorin ja luokittelija yhdistäminen GAN-verkoksi

Kuvion 20 rivillä kolme huomataan myös komento, jossa luokittelijamalli muokataan niin, että se ei ole koulutettavissa *gan-funktion* kautta, vaan koulutetaan itsenäisesti koulutussilmukassa luokittelijafunktion kautta. Generaattori sen sijaan koulutetaan *gan-funktion* kautta koulutussilmukassa. Tämän toiminnon ansiosta molemmat mallit saadaan koulutettua erikseen.

Kun GAN-verkko on valmis, voidaan alustaa se muuttujaan, jota käytetään verkon koulutuksessa. Lisäksi voidaan tarkastella verkon rakennetta tulostamalla se Kuvion 21 mukaisella tavalla. Kuviosta nähdään yksinkertaisen GAN-verkon rakenne: generaattorin syötekerros, generaattorin ulostulo ja luokittelijan syöte sekä ulostulona luokittelu datajoukon aitoudesta.



Kuvio 21. GAN-verkon rakenne

GAN-verkon koulutus tapahtuu koulutusfunktion kautta, joka saa parametreikseen: muuttujiin tallennetut GAN-verkon, generaattorin, luokittelijan, alkuperäisen datajoukon mitä pyritään mallintamaan, generaattorin syötteen ulottuvuuden, koulutusiterointien määrän, koulutusjoukon määrän ja aikavälin milloin koulutusta arvioidaan. Kuviosta 22 voidaan tarkastella GAN-verkon koulutusfunktiota tarkemmin rivinumeroiden avulla. Rivillä kolme määritellään koulutusjoukon koko, jolla koulutetaan luokittelijaa. Seuraavaksi riveillä kuusi ja seitsemän määritellään listat, johon tallennetaan koulutuksessa saadut häviöpalautteet. Häviöpalautteita tulostetaan koulutuksen aikana ja koulutuksen päätyttyä piirretään ne kuvaajaan.

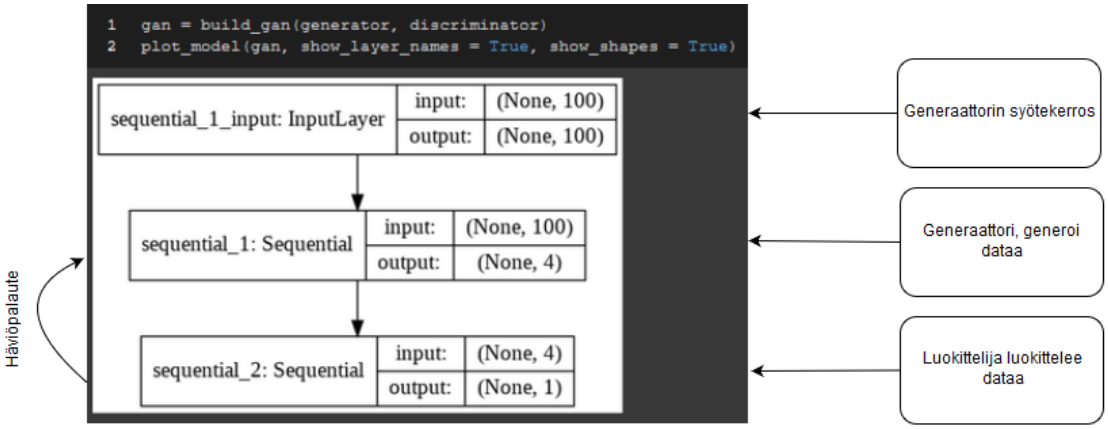
Riveillä 10-13 määritellään luokkamuuttujalistat, joita käytetään generaattorin ja luokittelijan koulutukseen. Luokkamuuttujilla kerrotaan luokittelija- ja generaattorimalleille koulutuksen yhteydessä, onko kyseessä alkuperäinen vai generoitu datajoukko (0 = generoitu data, 1 = alkuperäinen data).

```
1  def train(gan, generator, discriminator, data, latent_dim, n_epochs, n_batch, n_eval):
2      #Half batch size for updateting discriminator
3      half_batch = int(n_batch / 2)
4
5      #lists for stats from the model
6      generator_loss = []
7      discriminator_loss = []
8
9      #generate class labels for fake and real
10     valid = np.ones((half_batch, 1))
11     fake = np.zeros((half_batch, 1))
12     y_gan = np.ones((n_batch, 1))
13     #training loop
14     for i in range(n_epochs):
15
16         #select random batch from the real numerical data
17         idx = np.random.randint(0, categorical_data.shape[0], half_batch)
18         real_data = data[idx]
19
20         #generate fake samples from the noise
21         noise = np.random.normal(0, 1, (half_batch, latent_dim))
22         fake_data = generator.predict(noise)
23
24         #train the discriminator and return losses
25         d_loss_real, _ = discriminator.train_on_batch(real_data, valid)
26         d_loss_fake, _ = discriminator.train_on_batch(fake_data, fake)
27
28         d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
29         discriminator_loss.append(d_loss)
30
31         #generate noise for generator input and  train the generator (to have the discriminator label samples as valid)
32         noise = np.random.normal(0, 1, (half_batch, latent_dim))
33         g_loss = gan.train_on_batch(noise, valid)
34         generator_loss.append(g_loss)
35         #evaluate progress
36         if (i+1) % n_eval == 0:
37             print ("Epoch: %d [Generator loss: %f] [Discriminator loss: %f]" % (i + 1, g_loss, d_loss))
38     plt.figure(figsize = (20, 10))
39     plt.plot(generator_loss, label = "Generator loss")
40     plt.plot(discriminator_loss, label = "Discriminator loss")
41     plt.title("Stats from training GAN")
42     plt.legend()
```

Kuvio 22. GAN-verkon koulutus numeerisilla arvoilla

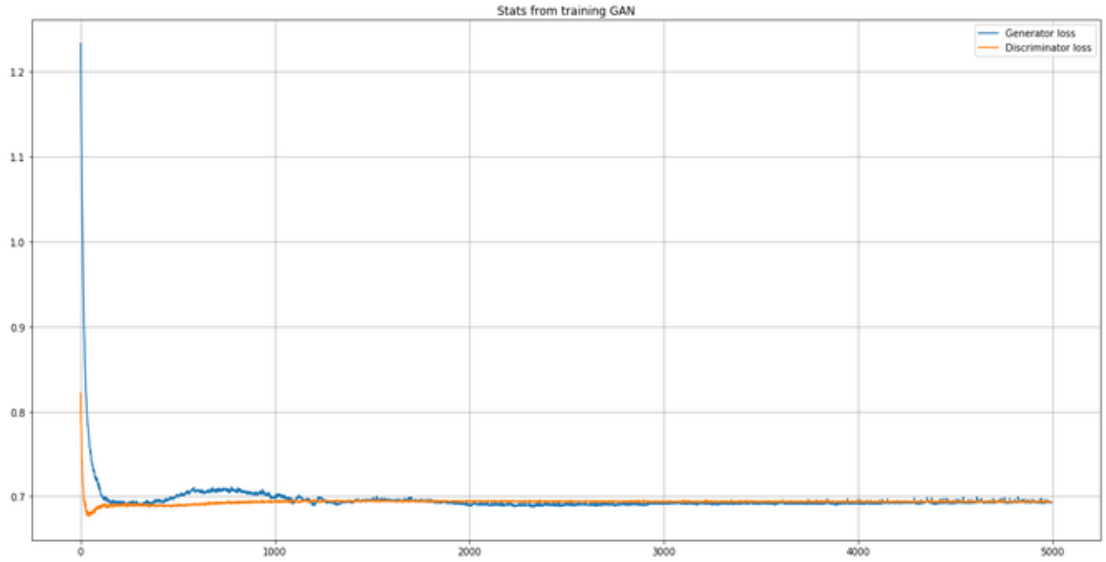
Riviltä 14 alkaen koodin loppuun saakka määritellään GAN-verkon koulutussilmukka. Koulutusfunktion syöte n_epochs määrittelee koulutuskertojen määrän GAN-verkon koulutuksessa. Silmukassa valitaan ensiksi satunnainen otos alkuperäisestä datasta ja tallennetaan se muuttujaan (riveillä 17-18). Tämän jälkeen luodaan ”kohinaa” eli generoidaan satunnainen otos normaalijakaumasta kuvion (rivillä 21), joka syötetään generaattorille, joka taas generoi uutta dataa, joka tallennetaan muuttujaan. Tämän jälkeen alkuperäinen sekä generoitu data ja niiden vastaavat luokkamuuttujat syötetään yksitellen luokittelijalle käyttäen Keras train_on_batch funktiota (riveillä 25-26). Funktio palauttaa häviöpalautteen luokittelijan koulutuksesta. Samalla luokittelijan painoja päivetetään häviöpalautteiden mukaan. Häviöpalautteen ansiosta luokittelija pystyy päivittämään painoja siten, että se tunnistaa onko kyseessä alkuperäinen vai generoitu datajoukko. Alussa luokittelija voi olla todella huono luokittelemaan sille syötettyä dataa, mutta lyhyen ajan kuluessa se oppii tunnistamaan niitä hyvin. Seuraavaksi lisätään luokittelijan koulutuksessa saatu häviöpalaute aikaisemmin luotuun listaan (riveillä 28-29), josta häviöpalautteita voidaan tulostaa koulutuksen edetessä sekä piirtää kuvaajalle koulutuksen loputtua.

Kun luokittelijalle on syötetty sekä alkuperäistä ja generoitua dataa, päivitetään generaattoria luodun gan-funktion kautta, jossa aikaisemmin oli määritelty luokittelijan painot vakioksi, eli luokittelijaa ei päivitetä gan-funktiossa. Generaattorille luodaan syöte satunnaisia muuttujia normaalijakaumasta (Kuvion 22 rivillä 32). Tämän jälkeen koulutetaan generaattoria gan-funktion kautta käyttäen Keras train_on_batch funktiota (rivillä 33). Kun generaattoria päivitetään kuvion 23 mukaisella tavalla luokittelijan virhepalautteen kautta, luokittelijalle kerrotaan, että generaattorin generoima data on alkuperäistä dataa (luokittelijaa ”huijataan”). Tämän toiminnan ansiosta luokittelijan virhepalaute on suuri ja näin saadaan generaattorista koulutuksen edetessä parempi (ks. luku 3.4).



Kuvio 23. Generaattorin päivitys gan-funktiossa

Koulutuksen aikana tulostetaan generaattorin ja luokittelijan virhepalautteet arviointiaikavälin mukaisesti (riveillä 36-37), koulutuksen lopuksi piirretään kuvaajaan nämä virhepalautteet kuvion 24 mukaisella tavalla. Numeerisen datan generointi voi tarkastella tarkemmin liitteen 1 sivuilta 3-11.



Kuvio 24. Numeerisen GAN-verkon häviöpalautteet

6.3.2 GAN-verkolla kategoristen muuttujien generointi

Kategorisen datan generointi tapahtuu lähes samalla tavalla kuin numeerisen datan generoinnissa. Ainoat pienet muutokset tapahtuvat *generaattorissa*, *luokittelijassa* ja *koulutusfunktiossa*. Generaattorifunktiossa vaihdetaan ulostulokerroksen solmujen määrää halutun kolumnimäärän mukaan sekä ulostulokerroksen aktivointifunktio muutetaan *softmax funktioksi* (ks. luku 3.3). Luokittelijafunktiosta vaihdetaan vain sen syöte vastaamaan generaattorin ulostuloa vastaavaksi. Lisäksi koulutusfunktiosta vaihdetaan vain sille syötettävä data, jota käytetään verkon koulutuksessa (ks. Liite 1, sivut 14-19).

6.3.3 GAN-verkolla kategoristen ja numeeristen muuttujien generointi

Kategoristen ja numeeristen muuttujien malli tehtiin käyttäen *Keras function API:a*, jonka avulla voidaan luoda monimutkikkaampi neuroverkko, kuten useampi ulostulo, syöte tai ns. oksia neuroverkolle. Kuviosta 25 nähdään, että generaattorifunktio saa syötteenä kolme parametriä, jotka ovat kategoriset muuttujat ja numeeriset muuttujat. Funktion parametrit kertovat kuinka monta kolumnia generaattorin tulisi generoida kyseisiä muuttujia. Kuviosta myös huomataan kuinka verkko alkaa muuttujasta

hidden_1, joka kuvastaa ensimmäistä piilotettua kerrosta. Seuraavaksi tulee hidden_2 kerros ja tästä eteenpäin aloitetaan erittelemään verkko kolmeen eri oksaan. Muuttuja branch_1_hidden_1 on ensimmäisen kategorisen muuttujan oksa, joka jatkuu aina muuttujaan branch_1_output muuttujaan asti, joka taas on kyseisen oksan ulostulo. Luomalla ns. oksia generaattorimallille pyritään käsittelemään numeeriset omassa oksassaan ja sekä kaikki kategoriset kolumnit (one hot enkoodatut) pyritään käsittelemään omissa oksissaan, jotta voidaan käyttää kategorisen datan generoinnissa ulostulokerroksessa softmax *aktivointifunktiota*. Lisäksi tällä metodilla saadaan oksat oppimaan paremmin sille syötetyn datan jakauman.

```
def build_generator(categorical_data_shape, categorical_data_shape2, numerical_data_shape):
    #noise as input from the latent space
    noise = Input(shape = (100,))
    hidden_1 = Dense(8, kernel_initializer = "he_uniform")(noise)
    hidden_1 = LeakyReLU(0.2)(hidden_1)
    hidden_1 = BatchNormalization(momentum = 0.8)(hidden_1)

    hidden_2 = Dense(16, kernel_initializer = "he_uniform")(hidden_1)
    hidden_2 = LeakyReLU(0.2)(hidden_2)
    hidden_2 = BatchNormalization(momentum = 0.8)(hidden_2)

    #Branch 1 for generating categorical gender data
    branch_1 = Dense(32, kernel_initializer = "he_uniform")(hidden_2)
    branch_1 = LeakyReLU(0.2)(branch_1)
    branch_1 = BatchNormalization(momentum = 0.8)(branch_1)

    branch_1 = Dense(64, kernel_initializer = "he_uniform")(branch_1)
    branch_1 = LeakyReLU(0.2)(branch_1)
    branch_1 = BatchNormalization(momentum=0.8)(branch_1)
    #Output 1 layer, softmax activation for multi classification
    branch_1_output = Dense(categorical_data_shape, activation = "softmax")(branch_1)

    #Branch 2 for generating categorical bmi_class data
    branch_2 = Dense(32, kernel_initializer = "he_uniform")(hidden_2)
    branch_2 = LeakyReLU(0.2)(branch_2)
    branch_2 = BatchNormalization(momentum=0.8)(branch_2)

    branch_2 = Dense(64, kernel_initializer = "he_uniform")(branch_2)
    branch_2 = LeakyReLU(0.2)(branch_2)
    branch_2 = BatchNormalization(momentum=0.8)(branch_2)
    #Output 2 layer, softmax activation for multi classification
    branch_2_output = Dense(categorical_data_shape2, activation = "softmax")(branch_2)

    #Branch 3 for generating numerical data
    branch_3 = Dense(64, kernel_initializer = "he_uniform")(hidden_2)
    branch_3 = LeakyReLU(0.2)(branch_3)
    branch_3 = BatchNormalization(momentum=0.8)(branch_3)

    branch_3 = Dense(128, kernel_initializer = "he_uniform")(branch_3)
    branch_3 = LeakyReLU(0.2)(branch_3)
    branch_3 = BatchNormalization(momentum=0.8)(branch_3)
    #Output 3, sigmoid activation
    branch_3_output = Dense(numerical_data_shape, activation = "sigmoid")(branch_3_hidden_2)

    #Combined output
    combined_output = concatenate([branch_1_output, branch_2_output, branch_3_output])
    #Return model
    return Model(inputs = noise, outputs = combined_output)
```

Kuvio 25. Kategoristen ja numeeristen muuttujien generaattori

Numeerisessa datan käsittelyssä käytetään viimeisessä ulostulokerroksessa sigmoid aktivointifunktiota. Lopuksi oksien ulostulot yhdistettiin yhdeksi ulostuloksi käyttäen Keras concatenate-funktiota. Oksien ulostulot yhdistetään, jotta saataisiin syötettyä generoitu data yhtenä taulukkona luokittelijalle, kuten myöskin alkuperäinen taulukko. Generaattorifunktio palauttaa mallin, jonka syötteenä on ”kohinaa” eli satunnaisia muuttujia latentista tilasta ja ulostulona on oksien yhdistetty ulostulo. Generaattorifunktion rakenne voidaan tulostaa hiukan helpommin ymmärrettävään muotoon Kuvion 26 mukaisella tavalla. Kuviosta 26 huomataan Keras function API:n hyödyt luoda useita oksia ja ulostuloja neuroverkolle.

Kuvio 26. Kategoristen ja numeeristen generaattorin rakenne

Kategoristen ja numeeristen muuttujien luokittelija on samanlainen kuin aikaisemmissa esimerkeissä, mutta se luodaan myös käyttäen Keras function API:a. Luokittelijafunktio saa syötteenä alkuperäisten kolumnien yhteenlasketun määrän kuvion 27 mukaisella tavalla.

```
def build_discriminator(inputs_n):
    #Input from generator
    d_input = Input(shape = (inputs_n,))
    d = Dense(128, kernel_initializer="he_uniform")(d_input)
    d = LeakyReLU(0.2)(d)
    d = Dense(64, kernel_initializer="he_uniform")(d)
    d = LeakyReLU(0.2)(d)
    d = Dense(32, kernel_initializer="he_uniform")(d)
    d = LeakyReLU(0.2)(d)
    d = Dense(16, kernel_initializer="he_uniform")(d)
    d = LeakyReLU(0.2)(d)
    d = Dense(8, kernel_initializer="he_uniform")(d)
    d = LeakyReLU(0.2)(d)
    #Discriminator output for classification, sigmoid activation
    d_output = Dense(1, activation = "sigmoid")(d)
    #compile and return model
    model = Model(inputs = d_input, outputs = d_output)
    model.compile(loss = "binary_crossentropy", optimizer = optimizer, metrics = ["accuracy"])
    return model
```

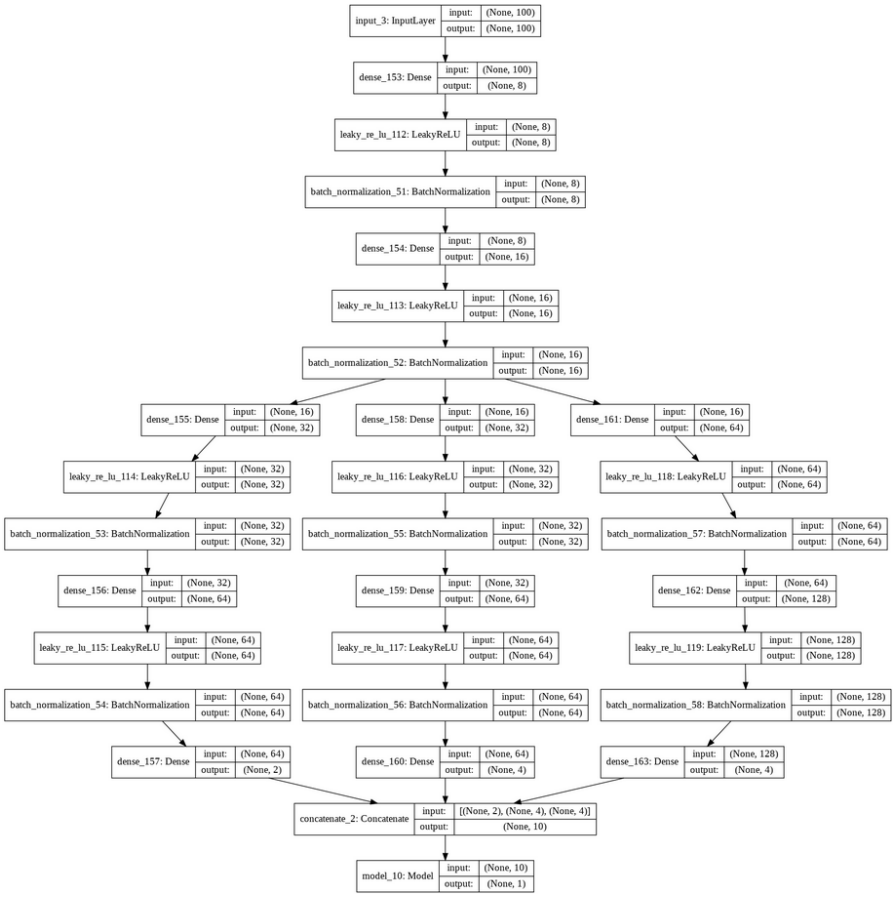
Kuvio 27. Kategorisen ja numeerisen datan luokittelijafunktio

GAN-verkon koostaminen tapahtuu myöskin käyttäen Keras function API:a, periaate on kuitenkin sama, kuin aikaisemmissa vaiheissa tapahtunut verkkojen yhdistäminen, eli yhdistetään generaattori ja luokittelija yhdeksi verkoksi kuvion 28 tapaan.

```
def build_gan(generator, discriminator):
    #Make discriminator not trainable
    discriminator.trainable = False
    #Discriminator takes input from generator and make discriminator GAN output
    gan_output = discriminator(generator.output)
    #Initialize gan
    model = Model(inputs = generator.input, outputs = gan_output)
    #Compile model
    model.compile(loss = "binary_crossentropy", optimizer = optimizer)
    #Return Model
    return model
```

Kuvio 28. GAN-verkko kategorisia ja numeerisia muuttujia varten

Gan-funktio saa syötteenä generaattorin ja luokittelija. Ensimmäiseksi luokittelijan painot muutetaan kouluttamattomaksi. Tämän jälkeen määritellään GAN-verkon ulostulo, joka tapahtuu yhdistämällä generaattorifunktion ulostulo luokittelija funktion syötteeksi. Sitten alustetaan GAN-verkko määrittelemällä sille syöte, jonka on generaattorifunktion syöte, eli satunnaisia muuttujia latentista tilasta ja ulostulona on aikaisemmin määritelty generaattorin ja luokittelijan yhdiste. Ulostulona toimii luokittelijan ulostulo eli luokittelu sille onko kyseinen datakehys generoitua -vai alkuperäistä dataa. Koko GAN-verkon rakenne voidaan tulostaa helpommin ymmärrettävään muotoon Kuvion 29 mukaisella tavalla.



Kuvio 29. Kategorinen ja numeerinen GAN-verkko

GAN-verkon koulutus tapahtuu samalla tavalla kuten aikaisemmissa esimerkeissä. Pienillä muokkauksilla voidaan toteuttaa GAN-verkon koulutus. Käytetään siis koulutusfunktiota kouluttamaan GAN-verkko, funktio saa syötteenä aikaisemmin määritellyt muuttujat Kuvion 30 mukaan: gan-funktio, generaattorifunktio, luokittelijafunktio, kategorinen data, toinen kategorinen data, numeerinen data, latentti ulottuvuus, harjoitussilmukoiden määrä, koulutuserän koko ja koulutuksen arviointiaikaväli.

```
def train(gan, generator, discriminator, categorical_data, categorical_data2, numerical_data,
latent_dim, n_epochs, n_batch, n_eval):
    #Half batch size for updateting discriminator
    half_batch = int(n_batch / 2)

    #lists for stats from the model
    discriminator_loss = []
    generator_loss = []

    #generate class labels for fake and real
    valid = np.ones((half_batch, 1))
    y_gan = np.ones((n_batch, 1))
    fake = np.zeros((half_batch, 1))
    #training loop
    for i in range(n_epochs):

        #select random batch from real categorical and numerical data
        idx = np.random.randint(0, categorical_data.shape[0], half_batch)
        gender_real = categorical_data[idx]
        bmi_real = categorical_data2[idx]
        numerical_real = numerical_data[idx]

        #concatenate categorical and numerical data for the discriminator
        real_data = np.concatenate([gender_real, bmi_real, numerical_real], axis = 1)

        #generate fake samples from the noise
        noise = np.random.normal(0, 1, (half_batch, latent_dim))
        fake_data = generator.predict(noise)

        #train the discriminator and return losses and acc
        d_loss_real, da_real = discriminator.train_on_batch(real_data, valid)
        d_loss_fake, da_fake = discriminator.train_on_batch(fake_data, fake)

        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
        discriminator_loss.append(d_loss)

        #generate noise for generator input and train the generator (to have the discriminator
label samples as valid)
        noise = np.random.normal(0, 1, (n_batch, latent_dim))
        g_loss = gan.train_on_batch(noise, y_gan)

        generator_loss.append(g_loss)
        #evaluate progress
        if (i+1) % n_eval == 0:
            print ("Epoch: %d [Discriminator loss: %f] [Generator loss: %f]" % (i + 1, d_loss,
g_loss))
    plt.figure(figsize = (20, 10))
    plt.plot(generator_loss, label = "Generator loss")
    plt.plot(discriminator_loss, label = "Discriminator loss")
    plt.title("Stats from training GAN")
    plt.grid()
    plt.legend()
```

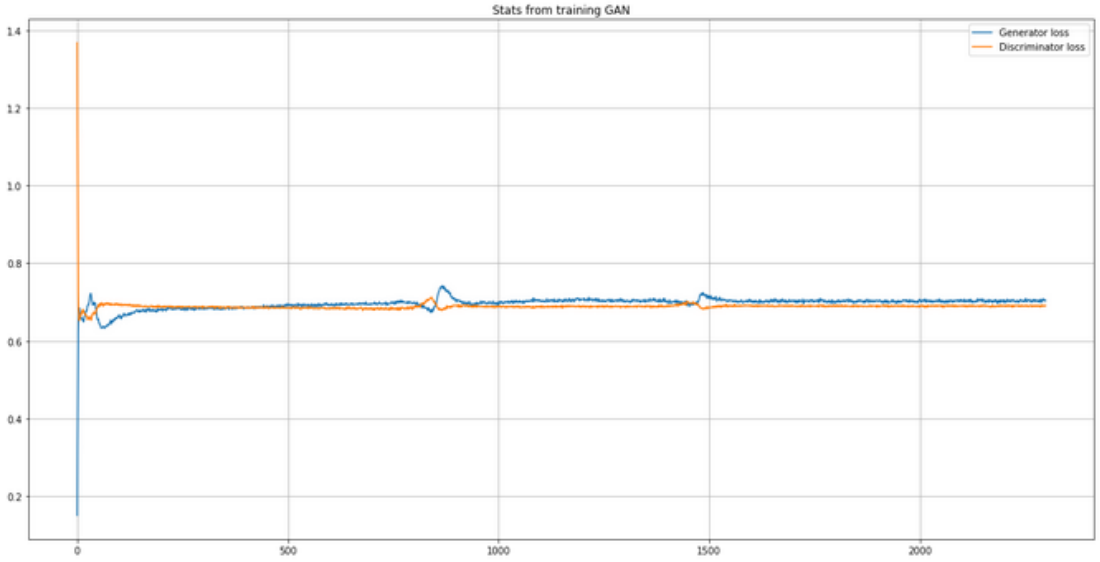
Kuvio 30. Kategorisen ja numeerisen GAN-verkon koulutus

GAN-verkon koulutuksessa ei tapahdu suuria muutoksia verraten aikaisempiin esimerkkeihin. Kuvion 31 mukaisella tavalla valitaan alkuperäisistä taulukoista satunnaiset otokset ja liitetään ne yhdeksi datakehykseksi.

```
#select random batch from real categorical and numerical data
idx = np.random.randint(0, categorical_data.shape[0], half_batch)
gender_real = categorical_data[idx]
bmi_real = categorical_data2[idx]
numerical_real = numerical_data[idx]

#concatenate categorical and numerical data for discriminator input
real_data = np.concatenate([gender_real, bmi_real, numerical_real], axis = 1)
```

Kuvio 31. Alkuperäisen datan valinta ja yhdistäminen

Koulutuksen aikana tulostetaan generaattorin ja luokittelijan virhepalautteet arviointiaikavälin mukaisesti, lopuksi piirretään kuvaajaan nämä virhepalautteet kuvion 32 mukaisella tavalla. Tätä esimerkkiä voi tarkastella tarkemmin liitteen 1 sivuilta 21-27.



Kuvio 32. Virhepalautteen kuvaaja

6.4 Uuden datan generointi ja sen muokkaus alkuperäiseen muotoon

GAN-verkkojen koulutuksen jälkeen voidaan käyttää eri malleja generoimaan niille määrättyjä datajoukkoja. Uuden datan generointi tapahtuu generaattorifunktiota käyttämällä. Keras kirjastossa neuroverkoille luodaan automaattisesti funktio, jonka avulla voidaan tehdä ennuste sille syötetyistä parametreistä. Kuten aikaisemmissa koulutusesimerkeissä luodaan ”kohinaa” generaattorin syötteeksi, suoritetaan sama toimenpide tässäkin Kuvion 33 mukaisella tavalla. Luodaan siis satunnaista kohinaa syötteeksi generaattorille. Lisäksi syötteessä määritellään kuinka monta riviä generaattori generoi ja kuinka monta kolumnia syötetään generaattorille (vastaava luku kuin koodissa määiritelty latent_dim).

```
noise = np.random.normal(0, 1, (10000, 100))
generated_mixed_data = generator4.predict(noise)
generated_mixed_data
```

```
array([[6.5177805e-09, 1.0000000e+00, 6.0876209e-01, ..., 6.1912340e-01,
        4.9177581e-01, 4.8557207e-01],
       [1.0913557e-24, 1.0000000e+00, 2.1655799e-18, ..., 5.6651312e-01,
        5.7311296e-01, 6.5712827e-01],
       [9.9999976e-01, 2.4986926e-07, 9.9999988e-01, ..., 3.1343400e-01,
        2.6918650e-01, 3.7169328e-01],
       ...,
       [1.0000000e+00, 8.3462080e-22, 1.0000000e+00, ..., 4.2388344e-01,
        3.3199945e-01, 4.4132009e-01],
       [1.1695668e-26, 1.0000000e+00, 3.8878457e-18, ..., 5.2725494e-01,
        5.7334667e-01, 7.0970201e-01],
       [2.0165229e-10, 1.0000000e+00, 3.3639376e-03, ..., 5.7177657e-01,
        4.4709170e-01, 4.0972564e-01]], dtype=float32)
```

Kuvio 33. Uuden datan generointi ja sen tuloste

Generaattori generoi uutta dataa, joka tulee numpy taulukko muodossa. Selvyyden vuoksi ja jatkotoimenpiteiden helpottamiseksi muokataan generoitu taulukko alkuperäiseen datakehys muotoon (ks. kuvio 34).

	Female	Male	healthy	overweight	severely overweight	underweight	patient_id	Height	Weight	bmi
0	0	1	0	0	1	0	90329	1.875714	109.721075	31.19
1	0	1	1	0	0	0	82793	1.747060	73.622792	24.12
2	0	1	0	1	0	0	98691	1.882397	96.497629	27.23
3	0	1	0	0	1	0	20430	1.821967	99.809586	30.07
4	0	1	0	0	1	0	96554	1.774998	93.598695	29.71
...	...	...	...	...	...	...	...	...	...	...
9995	1	0	1	0	0	0	51506	1.680785	62.041210	21.96
9996	1	0	0	1	0	0	38900	1.703506	77.504378	26.71
9997	1	0	1	0	0	0	26718	1.622247	58.275424	22.14
9998	1	0	1	0	0	0	67447	1.753470	74.322226	24.17
9999	1	0	1	0	0	0	70063	1.573384	51.550366	20.82

10000 rows × 10 columns

Kuvio 34. Alkuperäinen datakehys

Ensiksi siirrettään generoitu data datakehykseen ja lisätään siihen kolumnit. Kuten luvussa 5.3 todettiin, että generaattori generoi vain jatkuvia lukuja, joten uusi generoitu data sisältää vain jatkuvia lukuja ja kategoriset muuttujat muunnettiin pyöristämällä samanlaiseen muotoon kuin alkuperäinen (käyttäen numpy kirjastoa). Kuvioita 34 ja 35 vertaillessa huomataan, ettei neljä viimeisintä kolumnia (numeeriset luvut) ole samassa muodossa kuin alkuperäinen data.

```
columns = list(gender_ohe.columns) + list(bmi_class_ohe.columns) + list(numerical_data.columns)
mixed_gen_df = pd.DataFrame(data = generated_mixed_data, columns = columns)
mixed_gen_df
```

	Female	Male	healthy	overweight	severely overweight	underweight	patient_id	Height	Weight	bmi
0	2.169942e-22	1.000000e+00	9.060907e-28	1.000000e+00	1.399785e-16	7.472536e-15	0.391915	0.523241	0.547183	0.695682
1	0.000000e+00	1.000000e+00	1.194512e-23	1.000000e+00	4.207920e-14	3.176294e-13	0.900566	0.530419	0.531953	0.658244
2	1.142618e-34	1.000000e+00	3.077713e-24	1.000000e+00	9.867002e-15	1.067670e-12	0.279325	0.499410	0.469802	0.555910
3	4.545423e-25	1.000000e+00	5.989664e-15	2.749563e-27	1.000000e+00	8.026944e-11	0.266263	0.620275	0.701610	0.862301
4	1.076919e-30	1.000000e+00	1.094947e-23	1.000000e+00	1.229965e-15	1.832937e-14	0.098891	0.689789	0.688771	0.755700
...	...	...	...	...	...	...	...	...	...	...
9995	0.000000e+00	1.000000e+00	3.251181e-26	1.000000e+00	1.093360e-13	2.802349e-12	0.957873	0.751698	0.745320	0.769100
9996	1.000000e+00	7.142956e-28	1.000000e+00	0.000000e+00	1.858610e-27	7.697758e-28	0.204599	0.227191	0.243762	0.425217
9997	1.938128e-33	1.000000e+00	6.381097e-23	1.000000e+00	1.566945e-14	6.425635e-13	0.211781	0.771865	0.713186	0.672167
9998	3.656282e-25	1.000000e+00	2.889877e-16	9.999999e-01	2.409490e-08	1.605306e-07	0.501832	0.510715	0.493205	0.582027
9999	0.000000e+00	1.000000e+00	5.346916e-18	1.000000e+00	1.034068e-11	9.692935e-10	0.706568	0.484619	0.471114	0.578189

10000 rows × 10 columns

Kuvio 35. Generoitu data

Luvut ovat aikaisemmin *MinMaxScaler* luokalla skaalatussa muodossa. Luvut täytyy muuttaa takaisin alkuperäiseen muotoon käyttäen *MinMaxScaler inverse_transform-funktiota*. Näin saadaan kuvion 36 mukainen generoitu datakehys, joka on samaa muotoa kuin alkuperäinen.

```
mixed_gen_df.iloc[:, 0:6] = np.round(mixed_gen_df.iloc[:, 0:6])
mixed_gen_df.iloc[:, 6:10] = mms.inverse_transform(mixed_gen_df.iloc[:, 6:10])
mixed_gen_df
```

	Female	Male	healthy	overweight	severely overweight	underweight	patient_id	Height	Weight	bmi
0	0.0	1.0	0.0	1.0	0.0	0.0	39187.027344	1.707028	80.299980	27.680084
1	0.0	1.0	0.0	1.0	0.0	0.0	90042.507812	1.711537	78.881760	27.021935
2	0.0	1.0	0.0	1.0	0.0	0.0	27930.158203	1.692055	73.094398	25.222895
3	0.0	1.0	0.0	0.0	1.0	0.0	26624.236328	1.767992	94.679810	30.609245
4	0.0	1.0	0.0	1.0	0.0	0.0	9890.270508	1.811667	93.484344	28.735199
...	...	...	...	...	...	...	...	...	...	...
9995	0.0	1.0	0.0	1.0	0.0	0.0	95772.109375	1.850564	98.750061	28.970779
9996	1.0	0.0	1.0	0.0	0.0	0.0	20459.035156	1.521024	52.046021	22.925318
9997	0.0	1.0	0.0	1.0	0.0	0.0	21177.123047	1.863235	95.757790	27.266697
9998	0.0	1.0	0.0	1.0	0.0	0.0	50176.691406	1.699158	75.273636	25.682039
9999	0.0	1.0	0.0	1.0	0.0	0.0	70646.398438	1.682762	73.216553	25.614557

10000 rows × 10 columns

Kuvio 36. Generoitu data alkuperäisessä muodossa

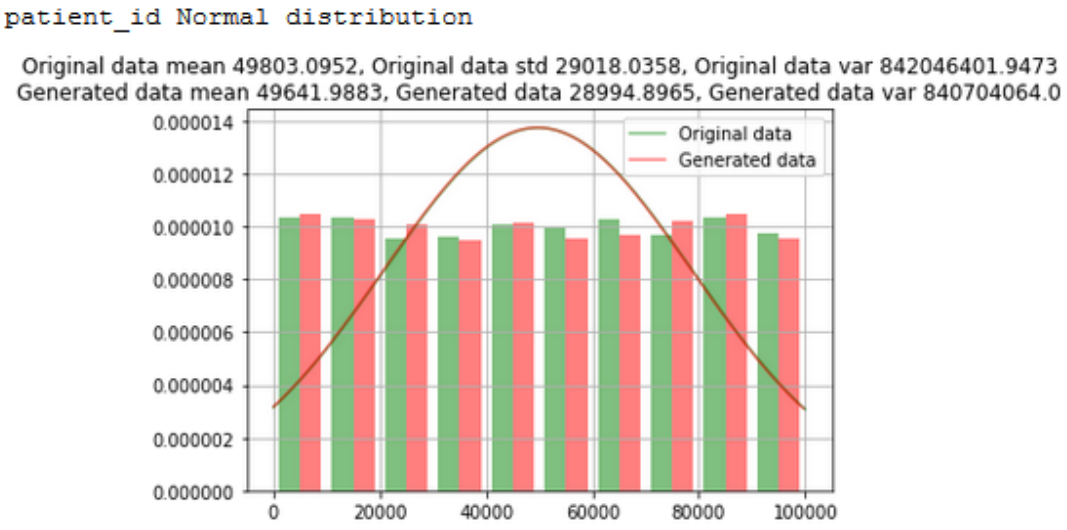
6.5 Generoidun datan ja alkuperäisen vertailu/visualisointi

Kun uusi generoitu data on muokattu alkuperäisen datakehyksen kanssa vastaavaan muotoon, voidaan aloittaa niiden vertailu keskenään. Ensiksi tarkastellaan alkuperäisen ja generoidun datan samankaltaisuutta, piirtämällä kuvaajaan normaalijakaumat generoidun ja alkuperäisen datakehyksen vastaavista kolumneista. Lisäksi tulostetaan molempien vastaavien kolumnien keskiarvot, keskihajonta sekä varianssi. Tätä varten luotiin funktio, joka suorittaa kyseiset toiminnot (ks. kuvio 37).

```
def normal_distribution(r, f):

        r_x = np.linspace(r.min(), r.max(), len(r))
        f_x = np.linspace(f.min(), f.max(), len(f))

        r_y = scipy.stats.norm.pdf(r_x, r.mean(), r.std())
        f_y = scipy.stats.norm.pdf(f_x, f.mean(), f.std())

        n, bins, patches = plt.hist([r, f], density = True, alpha = 0.5, color = ["green", "r
ed"])
        xmin, xmax = plt.xlim()

        plt.plot(r_x, r_y, color = "green", label = "Original data", alpha = 0.5)
        plt.plot(f_x, f_y, color = "red", label = "Generated data", alpha = 0.5)
        title = f"Original data mean {np.round(r.mean(), 4)}, Original data std {np.round(r.s
td(), 4)}, Original data var {np.round(r.var(), 4)}\nGenerated data mean {np.round(f.mean(),
4)}, Generated data {np.round(f.std(), 4)}, Generated data var {np.round(f.var(), 2)}"
        plt.title(title)
        plt.legend()
        plt.grid()
        plt.show()
```

Kuvio 37. Vertailufunktio

Funktio saa syötteenä alkuperäisen sekä generoidun data kolumnin. Funktio syötetään silmukkaan, joka käy yksitellen halutut kolumnit läpi ja tulostaa edellä mainitut vertailukohteet kuvion 38 mukaisella tavalla. Kuviosta voidaan helposti tarkastella alkuperäisen ja generoidun kolumnin eroja.

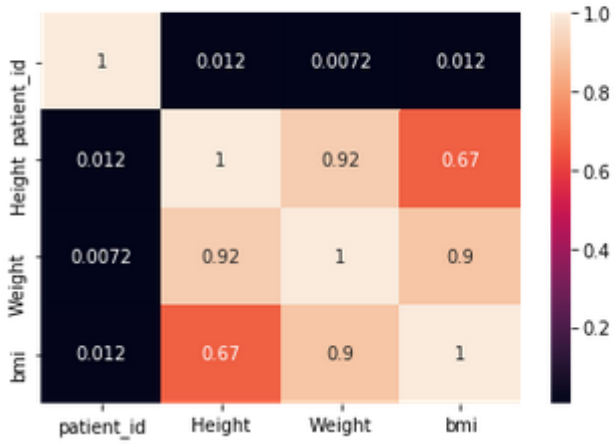


Kuvio 38. Vertailufunktion tuloste

Seuraavaksi tarkastellaan, onko generoidun datan muuttujien välillä samanlaista korrelaatiota, kuten alkuperäisessä datassa. Korrelaatioita on helppo tarkastella Python

Seaborn kirjaston avulla. Seaborn ja Pandas kirjastoilla voidaan luoda korrelaatiomatriisi kuvion 39 mukaisella tavalla. Sekä alkuperäinen data ja generoitu data sijoitetaan erillisille korrelaatiomatriiseille, josta niitä on helppo vertailla keskenään.

```
Generated data
<matplotlib.axes._subplots.AxesSubplot at 0x7fa3d4a59fd0>
```



Kuvio 39. Generoidun datan korrelaatiomatriisi

Seuraavaksi vertaillaan alkuperäisen sekä generoidun datakehyksen kategorisia muuttujia keskenään (one-hot enkoodatut muuttujat) laskemalla kuinka paljon niitä on. Muuttujien laskeminen käy helposti Kuvion 40 mukaisella tavalla (käyttämällä Pandas kirjaton value_counts funktiota) käymällä jokainen kolumni yksi kerrallaan läpi laskien kolumnissa olevat arvojen lukumäärän.

```
#original data value count
for column in og_data.iloc[:, 0:6].columns:
    print(og_data[column].value_counts())
```

```
1    5000
0    5000
Name: Gender_Female, dtype: int64
1    5000
0    5000
Name: Gender_Male, dtype: int64
0    5888
1    4112
Name: bmi_class_healthy, dtype: int64
1    5590
0    4410
Name: bmi_class_overweight, dtype: int64
0    9744
1     256
Name: bmi_class_severely overweight, dtype: int64
0    9958
1      42
Name: bmi_class_underweight, dtype: int64
```

Kuvio 40. Kategoristen arvojen laskeminen

Lopuksi käytetään neuroverkkoa luokittelemaan alkuperäistä sekä generoitua dataa. Koulutettuja neuroverkkoja voidaan arvioida kuinka hyvin ne luokittelevat niille syötettyä uutta dataa. Neuroverkko luotiin käyttäen Keras Sequential mallia kuvion 41 mukaisesti. Neuroverkkoja koulutetaan alkuperäisellä ja generoidulla datalla, jonka jälkeen niiden luokittelu tarkkuus arvioidaan syöttämällä niille dataa mitä ne ei ole aikaisemmin nähnyt. Testitulosten mukaan voidaan arvioida voiko generoitua dataa käyttää alkuperäisen datan sijaan.

```
def classifier():
    model = Sequential()
    model.add(Dense(128, activation = "relu", input_dim = 3))
    model.add(Dropout(0.2))
    model.add(Dense(64, activation="relu"))
    model.add(Dropout(0.2))
    model.add(Dense(32, activation="relu"))
    model.add(Dropout(0.2))
    model.add(Dense(16, activation="relu"))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation="sigmoid"))
    model.compile(optimizer = optimizer, loss = "binary_crossentropy", metrics = ["accurac
y"])
    return model

def classifier_train(classifier, x_train, y_train, epoch_limit=100, batch_size=256):

    history = classifier.fit(x_train, y_train, batch_size=batch_size, epochs=epoch_limit, ver
bose=0)

def evaluate(classifier, x_test, y_test):
    score = classifier.evaluate(x_test, y_test, verbose=1)
    print("Test Loss = {}, Test Accuracy = {}".format(score[0], score[1]))
```

Kuvio 41. Neuroverkkoluokittelija

Luokittelu tapahtuu siis jonkun kategorisen muuttujan mukaan. Ensiksi täytyy muokata data ominaisuuksiin ja ennustuksen kohteisiin. Kuviossa 42 käytetään sklearn kirjastosta löytyvää train_test_split funktiota, joka luo automaattisesti koulutus sekä testi dataa eri osioihin. Molempia neuroverkkoja koulutetaan ja testataan niiden omilla datakehyksillä (alkuperäinen ja generoitu data). Kun data on jaettu koulutus- sekä testidataksi, voidaan malli kouluttaa ja sen jälkeen testata miten hyvin malli suoriutuu uuden datan luokittelussa.

```
from sklearn.model_selection import train_test_split

X = og_data.iloc[:, 7:] #features
y = gender_labels #target labels
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size = 0.1, random_state = 42)
```

```
model = classifier()
classifier_train(model, x_train, y_train)
evaluate(model, x_test, y_test)
```

```
1000/1000 [==============================] - 4s 4ms/step
Test Loss = 0.22275649595260621, Test Accuracy = 0.915
```

Kuvio 42. Neuroverkkoluokittelijan koulutus ja arviointi

Kategoristen ja numeeristen datakehyksien vertailua/visualisointi voidaan tarkastella kokonaisuudessaan liitteen 1 sivuilta 30-37

7 Tutkimustulokset

Saadut tulokset numeerisen datan generoinnissa ovat hyvinkin lupaavia. Normaalijakauma, keskiarvo, keskihajonta ja varianssi ovat yllättävän samankaltaisia hyvinkin lyhyellä verkon koulutuksella. Korrelaatiot muuttujien välillä olivat hyvin samanlaisia, verraten alkuperäiseen dataan. (ks. Liite 1, sivut 12-14)

Kategoristen (one-hot enkoodatut) muuttujien generointi sen sijaan luo uusia haasteita. Kategorisen datan generoinnissa käytettiin hetero- ja homogeenistä datakehystä. Alkuperäistä ja generoitua datakehystä vertailtiin laskemalla kategoristen

muuttujien määrät. Homogeenisessä datakehyksessä oli vain kaksi kategorista muuttujaa, joita kumpaakin oli 5000 kappaletta. Generaattori generoi tällaisen datakehyksen keskiarvollisesti noin 0,1-1% virheellä, eli jompaakumpaa generoitua muuttujaa oli virhemäärän verran enemmän, tulokset olivat positiivisia. Suuria ongelmia tuli heterogeenisen datan generoinnissa, jossa myöskin datakehyksen rivimäärä oli 10000 ja siihen kuului neljä eri kategorista muuttujaa. Luokat ovat olivat hyvin epätasapainossa, yhteen luokaan kuului 5590 kappaletta muuttujia, toiseen luokkaan kuului 4112 kappaletta muuttujia, kolmanteen kuului 256 kappaletta muuttujia ja viimeiseen luokkaan kuului 42 kappaletta. Generaattori ei saanut generoitua pientä luokkaa 42 kappaletta (noin 0,5% kategorisista muuttujista) missään vaiheessa. Kahta suurinta luokaa generaattori generoi hyvin. Kolmatta luokkaa generaattori generoi kohtalaisesti, välillä hyvin. (ks. Liite 1, sivut 18-19 & 20)

Kategoristen ja numeeristen muuttujien generointi samanaikaisesti GAN-verkolla saatiin erittäin positiivisia tuloksi. Normaalijakauma, keskiarvo, keskihajonta ja varianssi olivat hyvin samanlaisia. Kategorisia muuttujia oli myös hyvin samanlainen määrä kuin alkuperäisessä datakehyksessä, kuitenkaan generaattori ei pystynyt generoimaan jo aiemmin todettua ongelmallista hyvin pientä määrää kategorisia muuttujia (lukuun ottamatta muutamaa kertaa). Korrelaatiot muuttujien välillä olivat hyvinkin samanlaisia verraten alkuperäiseen datakehykseen. Tämä implikoi sitä, että generoidulla datalla koulutettu neuroverkko suoriutuu yhtä hyvin luokittelussaan verraten alkuperäiseen datakehykseen. Seuraavaksi testattiin neuroverkolla, kuinka hyvin voidaan luokitella dataa alkuperäisellä ja generoidulla datalla. Neuroverkkoja koulutettiin luokittelemaan sille syötetyn datan perusteella, onko kyseessä nainen vai mies. Tuloksena saatiin hyvinkin positiivisia tuloksia, sillä generoidulla datalla koulutettu neuroverkko pystyi luokittelemaan dataa paremmin kuin alkuperäisellä datalla. Keskiarvollisesti generoitu data toimi paremmin luokittelussa kuin alkuperäisellä datalla koulutettu neuroverkko. (ks. Liite 1, sivut 31-37).

8 Pohdinta

8.1 Numeerisen datan generointi

Numeerista dataa generoivan GAN-verkon kehitys ei ole kovinkaan haastavaa, kun asiaan on hiukan perehtynyt. Verkko voi olla hyvinkin yksinkertainen (jopa paljon yksinkertaisempi kun tässä työssä esitelty verkon rakenne) eikä sitä ei tarvitse kouluttaa kuin hyvin lyhyt aika. Tulokset numeerisen datan generoinnissa ovat hyvin lupaavia ja positiivisia. Pienillä neuroverkon hyperparametrien hienosäätämisellä ja testailulla voitaisiin saavuttaa vieläkin samankaltaisempi generoitu data kuin alkuperäinen data. Kuten jo aiemmin todettua olivat vertailukohteet hyvin samankaltaisia alkuperäiseen datakehykseen nähden. Generoitua numeerista dataa voitaisiin käyttää lisäämällä alkuperäiseen datakehykseen generoitua dataa, täten saataisiin enemmän dataa, joka taas vahvistaa alkuperäisestä datasta löytyviä rakenteita.

8.2 Kategorisen datan generointi

Kategorista dataa generoivan GAN-verkon kehitys ei niin ikään ole kovin haastavaa. Generaattorifunktiosta vaihdetaan vain viimeisen kerroksen aktivointifunktio softmax funktioksi numeerisesta esimerkistä. Luokittelijaan vaihdetaan vain sen syötteen kokoa, muuten se pysyy samana. Homogeenisessä datajoukolla koulutettu GAN-verkko generoi hyvin homogeenistä dataa. Heterogeenisessä datajoukossa, jossa neljä erikokoista luokkamuuttujaa ja yksi luokkamuuttuja joukko sisälsi hyvin pienen määrä dataa koko datajoukosta (n. 0,5% koko datajoukosta) tuotti ongelmia. Generaattori ei generoinut kyseistä pientä määrää dataa missään vaiheessa. Tätä ongelmaa pyrittiin ratkaisemaan muuttamalla GAN-verkon rakennetta. Generaattoria ja luokittelijaa pyrittiin yksinkertaistamaan tai monimutkaistamaan paljonkin, mutta joka kerta generaattori jätti generoimatta kyseisen pienen datajoukon. Koulutussilmukoiden ja koulutusjoukon koon kasvattaminen tai pienentäminen ei myöskään tuottanut haluttua tulosta. Kyseistä ongelmaa voitaisiin lähteä ratkomaan valitsemalla alkuperäisestä datajoukosta tämä pieni määrä dataa, jota generaattori ei gene-

roinut ja kouluttaa uusi GAN-verkko tällä pienellä määrällä dataa. Koulutuksen jälkeen voidaan generoida lisää dataa, joka sitten lisättäisiin alkuperäiseen dataa, jota taas käytettäisiin kouluttamaan GAN-verkko täydennetyllä datalla.

8.3 Kategorisen ja numeerisen datan generointi

Kategoristen ja numeeristen muuttujien generointi samanaikaisesti tuotti aluksi päänvaivaa. Monien yrityksien jälkeen saatiin tuotettua erittäin positiivisia tuloksia, käyttäen Keras function API, jonka avulla voitiin luoda monimuotoisempi generaattori neuroverkko. Neuroverkko käsitteli kategoriset ja numeeriset luvut erikseen, jonka jälkeen ne liitettiin yhdeksi datajoukoksi. Erillisellä datan käsittelyllä saatiin generoidusta datasta hyvin samanlainen kuin alkuperäisestä. Vertailussa generoitu data oli hyvin samankaltainen kuin alkuperäinen ja kriteerit täyttyivät hyvin. Lopuksi testattiin alkuperäistä ja generoitua dataa erillisellä neuroverkolla, joka luokittelu datan kahden muuttujan (mies ja nainen) mukaan. Neuroverkko luokittelu generoitua dataa paremmin kuin alkuperäistä, joka kertoo taas siitä, että generoitu data oli jopa parempaa kuin alkuperäinen tähän ongelmaan. Tämä tulos oli myöskin erittäin positiivinen ja lupaava. Tässäkin esimerkissä ilmeni jo aiemmin todettu ongelma, hyvin pienen kategoristen muuttujien määrän generoinnissa. Tosin muutamalla kerralla generaattori sai generoitua kyseisiä muuttujia.

8.4 Yhteenveto

Tehtävänannossa oli hiukan ymmärtämisen vaikeuksia aiheen uutuuden ja monimutkaisuuden takia. Haluttu lopputulos muuttui hiukan työn edetessä. Lisäsin itse alkuperäisen ja generoidun datan väliseen vertailuun kriteerit korrelaatiomatriisi (näkyy muuttujien väliset korrelaatiot) ja luokitteleva neuroverkko, jonka avulla voitiin tarkastella generoidun datan käyttöä koneoppimisessa.

Tutkimuksesta opittiin perinteisen GAN-verkon toiminnasta ja sen rajoituksista. Kuten luvussa 5.3 mainittiin sekä tuloksista huomattiin, että GAN-verkko tuottaa vain jatkuvia lukuja. Työssä pyöristettiin kategoriset luvut, sekä joitakin numeerisia lukuja

samaan muotoon kuin alkuperäinen data. Ilmeisesti luvussa 4 mainitulla variaatio autoenkooderilla voidaan generoida vain disrkeettejä lukuja. Varitaatio autoenkooderi voitaisiin lisätä GAN-verkkoon generoimaan diskreettejä lukuja, mutta se vaatiikin jo uuden projektin, eikä tähän aiheeseen perehdytty sen enempää.

Myöskin testattiin pienemmällä otannalla alkuperäisestä datasta, miten hyvin GAN-verkko toimii vähällä määrällä dataa. Huomattiin ettei datan määrällä ollut juuri vaikutusta generoidun datan laatuun verraten alkuperäiseen dataan. Kuitenkin huomattiin samojen ongelmien ilmenevän epätasapainoisen datan generoinnissa.

Mielestäni tutkimuksessa saavutettiin hyvin kategoristen ja numeeristen generoinnissa haluttu lopputulos. Kuten luvussa 7 todettiin, oli halutut vertailukohteet alkuperäisen ja generoidun datan kohdalla hyvinkin samanlaisia. Generoidulla datalla voisi olla käyttöä datan lisäyksessä, jos alkuperäinen datajoukko on vajavainen. Dataa lisätessä alkuperäiseen datajoukkoon voitaisiin käyttää esimerkiksi koneoppimisen malleissa vahvistamaan ennustustarkkuutta. Lisäksi generoitua dataa voitaisiin käyttää korvaamaan arkaluontoista dataa kuten terveysdataa (ainakin työssä testatun leludatasetin mukaan). Voidaan todeta, että GAN-verkolla voidaan hyvin helposti generoida homogeenistä dataa alkuperäisen pohjalta ja se oppii hyvin rivien todennäköisyysjakauman. Sen sijaan heterogeeninen data tuottaa ongelmia ja tällainen hyvin epätasapainoinen data on todella yleistä reaalimaailmassa. GAN-verkko on myöskin hyvin sattumanvarainen, johtuen generaattorin syötteen olevan satunnainen otos latentista tilasta. Tästä seuraa vaikeuksia löytää optimaalisin koulutus GAN-verkolle, sillä jokainen koulutus on erilainen ja tulokset vaihtuvat joka kerta. Myöskin koulutuksen jälkeen generaattorilla generoitu data muuttuu joka kerta satunnaisen syötteen ansiosta.

Työ oli mielestäni hyvin haasteellinen sen laajuuden ja monimuotoisuuden takia. Ensiksi tuli perehtyä GAN-verkon toimintaan, joka ei ollut millään tavalla tuttu entuudestaan. Aiheesta löytyi hyvin paljon tutkimuspapereita, joiden lukeminen aluksi tuotti vaikeuksia, eikä niistä saanut juuri mitään irti. Aiheeseen enemmän perehdyttyä niistä oli hyötyä. Myöskin GAN-verkoista on tehty satoja eri versiota erilaisiin käyttötarkoituksiin, joka lisäsi myös haastavuutta, koska oli vaikeaa selvitellä miten

kannattaisi lähestyä työn ongelmaa noviisina aiheen saralla. Työssä myöskin täytyi itse joko internetistä tai tehdä itse alkuperäinen datajoukko, joka soveltuisi työhön. Tämäkin tuotti suuria vaikeuksia, koska oli pientä epäselvyyttä sen suhteen mitä generoidulla datalla haluttiin tehdä. Myöskin datajoukon valinnan suhteen vaikutti se seikka miten hyvin alkuperäinen datajoukko kuvastaa reaalimaailmaa. Työssä myöskin suoritettiin monia muita vaiheita kuten datan muokkaus neuroverkolle ymmärrettävään muotoon, generoidun datan tulkitseminen, datan visualisointia, erilaiset neuroverkon funktiot ja datan vertailu. Nämä vaiheet olivat entuudestaan jo tuttuja minulle, joka helpotti työn tekemistä. Työssä ei perehdytty juurikaan GAN-verkon teoriaa, sillä toimivimpia ratkaisuja haettiin empiirisillä tutkimuksilla. Lisäksi internetin kirjallisuuden mukaan GAN-verkon kehityksestä on suoritettu paljon empiirisillä tutkimisella.

Työssä lähdettiin myös ratkaisemaan heterogeenisen datan generoinnissa ilmennyttä ongelmaa luvussa 5.3 mainitulla CGAN:illa (ks. Liite 1, sivut 38-49). CGAN:illa pyrittiin generoimaan painoindeksiluokan, sekä sukupuolen perusteella dataa, mutta sitä ei saatu toimimaan halutulla tavalla. Molempia ongelmia myöskin pyrittiin ratkaisemaan käyttäen ACGAN:ia, mutta sitä osiota ei lisätty työhön, sen puutteellisuuden takia. Lisäksi testattiin erilaisia GAN-verkkoja kuten WGAN, BGAN sekä muita omia yrityksiä generoida dataa, mutta päädyttiin käyttämään yksinkertaista GAN-verkkoa numeerisen ja kategorisen datan generoinnissa. Esittelin työni koodausosion sekä niiden tulokset kolmelle Tiedon asiantuntioille, ja kaikki kertoivat koodin sekä tuloksien olevan erittäin positiivisia. Myöskin päädyttiin siihen ratkaisuun, ettei epätasapainoisen datan generoinnissa ilmennyttä ongelmaa ole minun tarpeellista lähteä ratkomaan tässä työssä enää.

Lähteet

Bengio, Y., Courville, A., Goodfellow, I., Mirza, M., Ozair, S., Pouget-Abadie, J., Warde-Farley, D. & Xu, B. 2014. Generative Adversarial Nets. Tutkimuspaperi. Viitattu 31.10.2019. https://arxiv.org/pdf/1406.2661.pdf

Brownlee, J. 2019a. A Gentle Introduction to Generative Adversarial Networks (GANs). Koneoppimisen opetussivusto. Viitattu 30.10.2019. https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

Brownlee, J. 2019b. How to identify and Diagnose GAN Failure Modes. Koneoppimisen opetussivusto. Viitattu 11.7.2019. https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/

Deep Blue N.d. Ibm Deep Blue. Verkkosivu. Viitattu 23.10.2019 https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

Gan. N.d. Generative adversarial networks. Googlen koneoppimisen opetussivusto. Viitattu 31.10.2019 https://developers.google.com/machine-learning/gan

Goodfellow, I. 2016. NIPS 2016 Tutorial: Generative Adversarial Networks. Yleiskatsaus. Viitattu 11.7.2019 https://arxiv.org/pdf/1701.00160.pdf

Hong, Y., Hwang, U., Yoo, J., Yoon & Yoon, S. 2019. How Generative Adversarial Networks and Their Variants Work: An Overview. Yleiskatsaus. Viitattu 5.11.2019. https://arxiv.org/pdf/1711.05914.pdf

Kapoor, A. 2019. Deep Learning vs. Machine Learning: A Simple Explanation. Verkkoartikkeli. Viitattu 28.10.2019. https://hackernoon.com/deep-learning-vs-machine-learning-a-simple-explanation-47405b3eef08

Khosgoftaar, T & Shorten, C. 2019. A survey on Image Data Augmentation for deep learning. Yleiskatsaus. Viitattu 6.11.2019. https://link.springer.com/content/pdf/10.1186%2Fs40537-019-0197-0.pdf

Marr, B. 2019. A Short History of Deep Learning – Everyone Should Read. Verkkoartikkeli. Viitattu 29.10.2019. https://www.forbes.com/sites/bernardmarr/2016/03/22/a-short-history-of-deep-learning-everyone-should-read/#1fcfc2735561

Massaron, L. & Mueller, J. 2018. Artificial intelligence for dummies. New Jersey: John Wiley & Sons.

Massaron, L. & Mueller, J. 2019. Deep Learning for Dummies. New Jersey: John Wiley & Sons.

Meistä N.d. Toimeksiantajan verkkosivu. Viitattu 10.10.2019 https://www.tieto.com/fi/meista/tieto-yrityksena/

Multi-Class Neural Networks: Softmax. N.d. Artikkeli Google sivuilla. Viitattu 7.11.2019. https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax

Nicholson, C. N.d. A Beginner Guide to Backpropagation in Neural Network. Verkkoartikkeli. Viitattu 30.10.2019. https://skymind.ai/wiki/generative-adversarial-network-gan

Shaw, R. 2019. The 10 Best Machine Learning Algorithms for Data Science Beginners. Verkkoartikkeli. Viitattu 30.10.2019. https://www.dataquest.io/blog/top-10-machine-learning-algorithms-for-beginners/

Train, Test & Validation Sets explained. 2017. Machine Learning & Deep Learning Fundamentals. Kone- ja syväoppimisen opetussivu. Viitattu 25.10.2019. https://deeplizard.com/learn/video/Zi-0rlM4RDs

Underfitting in a Neural Network explained. 2017. Machine Learning & Deep Learning Fundamentals. Kone- ja syväoppimisen opetussivu. Viitattu 22.10.2019. https://deeplizard.com/learn/video/0h8lAm5Ki5g

Yu, A. 2019. How Netflix uses Ai, Data Science, and Machine Learning – From A Product Perspective. Verkkoartikkeli. Viitattu 28.10.2019. https://becominghuman.ai/how-netflix-uses-ai-and-machine-learning-a087614630fe

Liitteet

Liite 1. GAN

GAN

In [0]:
```
#imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

import tensorflow as tf
%tensorflow_version 1.x
print(tf.__version__)

import warnings
warnings.filterwarnings("ignore")

from keras.layers import Input, Dense, Reshape, Flatten, Dropout, BatchNormalization, Embedding
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.merge import concatenate
from keras.models import Sequential, Model
from keras.optimizers import Adam
from keras.utils import to_categorical
from keras.layers.advanced_activations import LeakyReLU
from keras.utils.vis_utils import plot_model
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder, LabelEncoder
import scipy.stats
```

The default version of TensorFlow in Colab will soon switch to TensorFlow 2.x.
We recommend you upgrade (https://www.tensorflow.org/guide/migrate) now or ensure your notebook will continue to use TensorFlow 1.x via the `%tensorflow_version 1.x` magic: more info (https://colab.research.google.com/notebooks/tensorflow_version.ipynb).

```
1.15.0

Using TensorFlow backend.
```

Dataframe

In [0]:
```
df = pd.read_csv("https://raw.githubusercontent.com/im-p/synteettinen_data/master/bmi_data_events.csv")
df
```

Out[0]:

	patient_id	Gender	Height	Weight	bmi	bmi_class	2017	2018	2019	2020
0	90329	Male	1.875714	109.721075	31.19	severely overweight	laakarikaynti 2	laakarikaynti 4	laakarikaynti 2	laakarikaynti 1
1	82793	Male	1.747060	73.622792	24.12	healthy	ei jatkotoimenpiteita	laakarikaynti 1	ei jatkotoimenpiteita	ei jatkotoimenpiteita
2	98691	Male	1.882397	96.497629	27.23	overweight	laakarikaynti 2	laakarikaynti 1	laakarikaynti 1	laakarikaynti 2
3	20430	Male	1.821967	99.809586	30.07	severely overweight	laakarikaynti 1	ei jatkotoimenpiteita	laakarikaynti 2	ei jatkotoimenpiteita
4	96554	Male	1.774998	93.598695	29.71	overweight	laakarikaynti 4	laakarikaynti 4	laakarikaynti 4	laakarikaynti 3
...	...	...	...	...	...	...	...	...	...	...
9995	51506	Female	1.680785	62.041210	21.96	healthy	ei jatkotoimenpiteita	ei jatkotoimenpiteita	laakarikaynti 1	ei jatkotoimenpiteita
9996	38900	Female	1.703506	77.504378	26.71	overweight	laakarikaynti 3	laakarikaynti 2	laakarikaynti 4	ei jatkotoimenpiteita
9997	26718	Female	1.622247	58.275424	22.14	healthy	ei jatkotoimenpiteita	ei jatkotoimenpiteita	ei jatkotoimenpiteita	laakarikaynti 1
9998	67447	Female	1.753470	74.322226	24.17	healthy	ei jatkotoimenpiteita	laakarikaynti 1	laakarikaynti 1	ei jatkotoimenpiteita
9999	70063	Female	1.573384	51.550366	20.82	healthy	ei jatkotoimenpiteita	ei jatkotoimenpiteita	ei jatkotoimenpiteita	ei jatkotoimenpiteita

10000 rows × 10 columns

Categorical and numerical data

In [0]:
```
numerical_data = df.select_dtypes("number")
categorical_data = df.select_dtypes("object")
numerical_data
```

Out[0]:

	patient_id	Height	Weight	bmi
0	90329	1.875714	109.721075	31.19
1	82793	1.747060	73.622792	24.12
2	98691	1.882397	96.497629	27.23
3	20430	1.821967	99.809586	30.07
4	96554	1.774998	93.598695	29.71
...	...	...	...	...
9995	51506	1.680785	62.041210	21.96
9996	38900	1.703506	77.504378	26.71
9997	26718	1.622247	58.275424	22.14
9998	67447	1.753470	74.322226	24.17
9999	70063	1.573384	51.550366	20.82

10000 rows × 4 columns

In [0]:
```
categorical_data
```

Out[0]:

	Gender	bmi_class	2017	2018	2019	2020
0	Male	severely overweight	laakarikaynti 2	laakarikaynti 4	laakarikaynti 2	laakarikaynti 1
1	Male	healthy	ei jatkotoimenpiteita	laakarikaynti 1	ei jatkotoimenpiteita	ei jatkotoimenpiteita
2	Male	overweight	laakarikaynti 2	laakarikaynti 1	laakarikaynti 1	laakarikaynti 2
3	Male	severely overweight	laakarikaynti 1	ei jatkotoimenpiteita	laakarikaynti 2	ei jatkotoimenpiteita
4	Male	overweight	laakarikaynti 4	laakarikaynti 4	laakarikaynti 4	laakarikaynti 3
...	...	...	...	...	...	...
9995	Female	healthy	ei jatkotoimenpiteita	ei jatkotoimenpiteita	laakarikaynti 1	ei jatkotoimenpiteita
9996	Female	overweight	laakarikaynti 3	laakarikaynti 2	laakarikaynti 4	ei jatkotoimenpiteita
9997	Female	healthy	ei jatkotoimenpiteita	ei jatkotoimenpiteita	ei jatkotoimenpiteita	laakarikaynti 1
9998	Female	healthy	ei jatkotoimenpiteita	laakarikaynti 1	laakarikaynti 1	ei jatkotoimenpiteita
9999	Female	healthy	ei jatkotoimenpiteita	ei jatkotoimenpiteita	ei jatkotoimenpiteita	ei jatkotoimenpiteita

10000 rows × 6 columns

Data preprocessing

numerical data

In [0]:
```
#Rescaleing data between 0-1
mms = MinMaxScaler()
numerical_data_rescaled = mms.fit_transform(numerical_data)
numerical_data_rescaled
```

Out[0]:
```
array([[0.90343165, 0.79172838, 0.863139  , 0.89533561],
       [0.82805733, 0.58695829, 0.4754764 , 0.49317406],
       [0.98706754, 0.8023644 , 0.72113127, 0.67007964],
       ...,
       [0.26720077, 0.38830089, 0.31065968, 0.38054608],
       [0.67456817, 0.59715974, 0.48298768, 0.4960182 ],
       [0.70073314, 0.31052854, 0.23843869, 0.30546075]])
```

Categorical data

In [0]:
```
ohe_data = pd.get_dummies(categorical_data)
ohe_data
```

Out[0]:

	Gender_Female	Gender_Male	bmi_class_healthy	bmi_class_overweight	bmi_class_severely overweight	bmi_class_underweight	2017_ei jatkotoimenpiteita	20
0	0	1	0	0	1	0	0	
1	0	1	1	0	0	0	1	
2	0	1	0	1	0	0	0	
3	0	1	0	0	1	0	0	
4	0	1	0	1	0	0	0	
...	...	...	...	...	...	...	...	
9995	1	0	1	0	0	0	1	
9996	1	0	0	1	0	0	0	
9997	1	0	1	0	0	0	1	
9998	1	0	1	0	0	0	1	
9999	1	0	1	0	0	0	1	

10000 rows × 26 columns

In [0]:
```
gender_ohe = ohe_data.iloc[:, 0:2]
bmi_class_ohe = ohe_data.iloc[:, 2:6]
events_ohe = ohe_data.iloc[:, 6:]
```

In [0]:
```
print("numerical data shape:", numerical_data_rescaled.shape)
print("gender ohe data shape:", gender_ohe.shape)
print("bmi class ohe data shape:", bmi_class_ohe.shape)
print("events ohe data shape:", events_ohe.shape)
```

```
numerical data shape: (10000, 4)
gender ohe data shape: (10000, 2)
bmi class ohe data shape: (10000, 4)
events ohe data shape: (10000, 20)
```

GAN for generating numerical data

Generator

- Generator takes input from the random normal distribution (latent_dim)
- Output the same number of columns as the original (n_columns)
- Generator will not be compiled, because we update generator based on the discriminators error

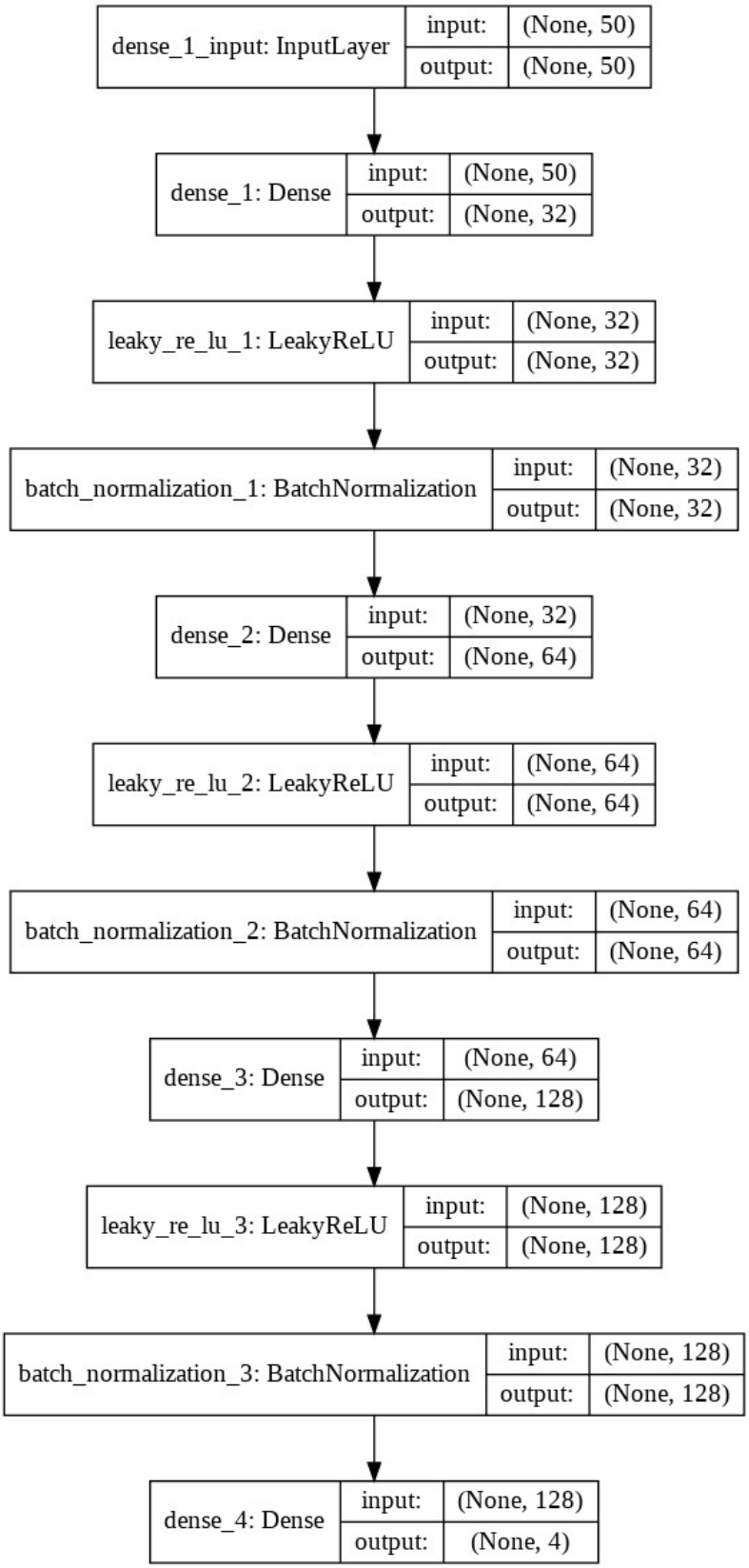
In [0]:
```
def build_generator(n_columns, latent_dim):
    model = Sequential()
    model.add(Dense(32, kernel_initializer = "he_uniform", input_dim=latent_dim))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(64,  kernel_initializer = "he_uniform"))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(128,  kernel_initializer = "he_uniform"))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(n_columns, activation = "sigmoid"))
    return model
```

Initialize generator and plot model

In [0]:
```
latent_dim = 50
generator = build_generator(numerical_data_rescaled.shape[1], latent_dim)
plot_model(generator, show_layer_names = True, show_shapes = True)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:66: Th
e name tf.get_default_graph is deprecated. Please use tf.compat.v1.get_default_graph instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:541: T
he name tf.placeholder is deprecated. Please use tf.compat.v1.placeholder instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:4432:
The name tf.random_uniform is deprecated. Please use tf.random.uniform instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:148: T
he name tf.placeholder_with_default is deprecated. Please use tf.compat.v1.placeholder_with_default inste
ad.
```

Out[0]:



Discriminator

- Dicscriminator input is the generator output
- Outputs classification (real or generated data)

In [0]:
```
optimizer = Adam(lr=0.0002, beta_1=0.5)
```

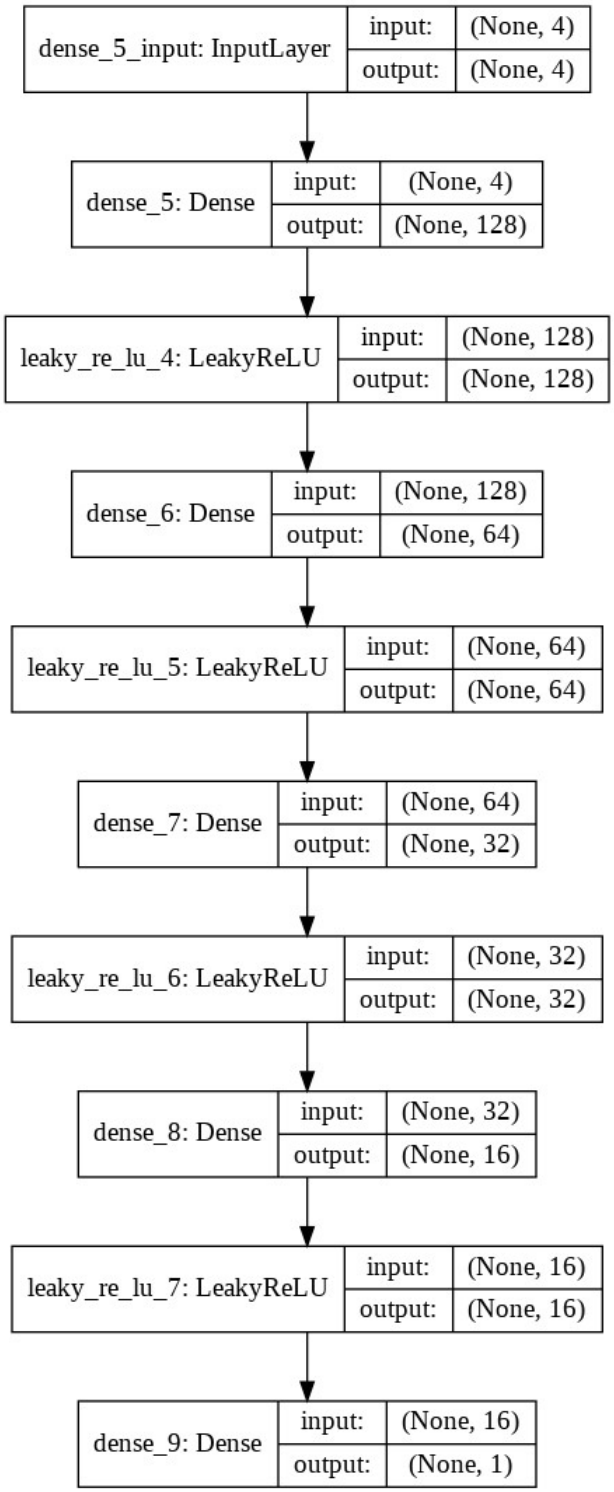
In [0]:
```
def build_discriminator(inputs_n):
        model = Sequential()
        model.add(Dense(128,  kernel_initializer = "he_uniform", input_dim = inputs_n))
        model.add(LeakyReLU(0.2))
        model.add(Dense(64,  kernel_initializer = "he_uniform"))
        model.add(LeakyReLU(0.2))
        model.add(Dense(32,  kernel_initializer = "he_uniform"))
        model.add(LeakyReLU(0.2))
        model.add(Dense(16,  kernel_initializer = "he_uniform"))
        model.add(LeakyReLU(0.2))
        model.add(Dense(1, activation = "sigmoid"))
        model.compile(loss = "binary_crossentropy", optimizer = optimizer, metrics = ["accuracy"])
        return model
```

Initialize discriminator and plot model

In [0]:
```
discriminator = build_discriminator(numerical_data_rescaled.shape[1])
plot_model(discriminator, show_layer_names = True, show_shapes = True)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/optimizers.py:793: The name tf.trai
n.Optimizer is deprecated. Please use tf.compat.v1.train.Optimizer instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:3657:
The name tf.log is deprecated. Please use tf.math.log instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow_core/python/ops/nn_impl.py:183:
where (from tensorflow.python.ops.array_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
```

Out[0]:



GAN

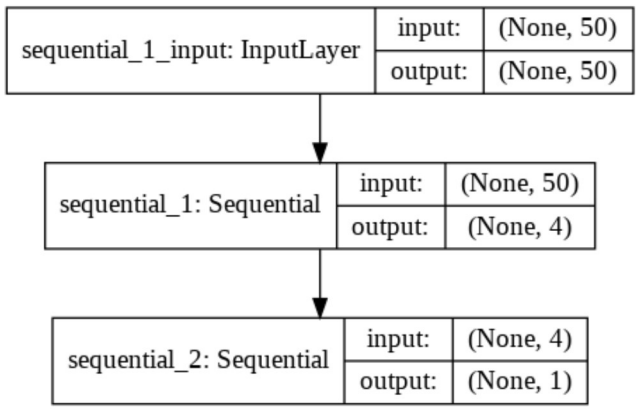
- Satck generator and discriminator
- GAN input is generator input
- GAN output classification generated or original data (discriminator output)
- We use gan function to train generator in training loop based on the classification error

In [0]:
```
def build_gan(generator, discriminator):
        # make weights in the discriminator not trainable
        discriminator.trainable = False
        # connect generator and dicriminator
        model = Sequential()
        # add generator
        model.add(generator)
        # add the discriminator
        model.add(discriminator)
        # compile model
        model.compile(loss = "binary_crossentropy", optimizer = optimizer)
        return model
```

Compile GAN and plot model

In [0]:
```
gan = build_gan(generator, discriminator)
plot_model(gan, show_layer_names = True, show_shapes = True)
```

Out[0]:

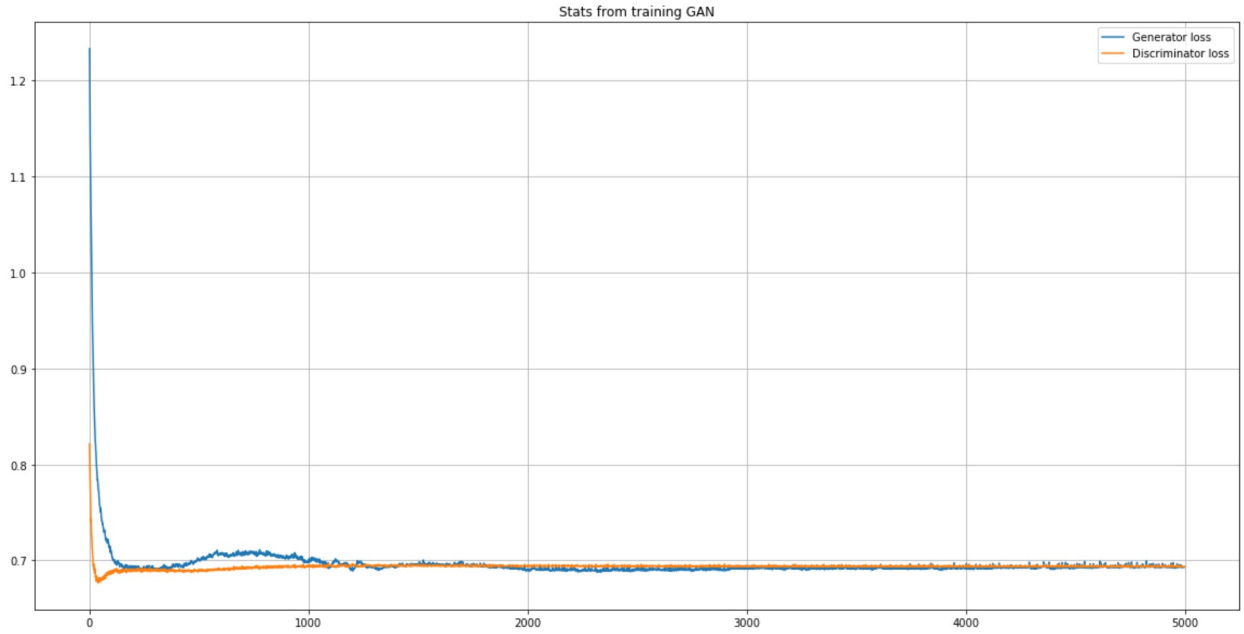


GAN training

In [0]:
```
def train(gan, generator, discriminator, data, latent_dim, n_epochs, n_batch, n_eval):
    #Half batch size for updateting discriminator
    half_batch = int(n_batch / 2)

    #lists for stats from the model
    generator_loss = []
    discriminator_loss = []

    #generate class labels for fake = 0 and real = 1
    valid = np.ones((half_batch, 1))
    fake = np.zeros((half_batch, 1))
    y_gan = np.ones((n_batch, 1))
    #training loop
    for i in range(n_epochs):

        #select random batch from the real numerical data
        idx = np.random.randint(0, data.shape[0], half_batch)
        real_data = data[idx]

        #generate fake samples from the noise
        noise = np.random.normal(0, 1, (half_batch, latent_dim))
        fake_data = generator.predict(noise)

        #train the discriminator and return losses
        d_loss_real, _ = discriminator.train_on_batch(real_data, valid)
        d_loss_fake, _ = discriminator.train_on_batch(fake_data, fake)

        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
        discriminator_loss.append(d_loss)

        #generate noise for generator input and  train the generator (to have the discriminator label sampl
es as valid)
        noise = np.random.normal(0, 1, (n_batch, latent_dim))
        g_loss = gan.train_on_batch(noise, y_gan)
        generator_loss.append(g_loss)

        #evaluate progress
        if (i+1) % n_eval == 0:
            print ("Epoch: %d [Generator loss: %f] [Discriminator loss: %f]" % (i + 1, g_loss, d_loss))

    #plot losses after training
    plt.figure(figsize = (20, 10))
    plt.plot(generator_loss, label = "Generator loss")
    plt.plot(discriminator_loss, label = "Discriminator loss")
    plt.title("Stats from training GAN")
    plt.legend()
    plt.grid()
```

In [0]:
```
train(gan, generator, discriminator, numerical_data_rescaled, latent_dim, n_epochs = 5000, n_batch = 1024,
n_eval = 250)
```

```
Epoch: 250 [Generator loss: 0.692752] [Discriminator loss: 0.689638]
Epoch: 500 [Generator loss: 0.700666] [Discriminator loss: 0.690210]
Epoch: 750 [Generator loss: 0.707478] [Discriminator loss: 0.692872]
Epoch: 1000 [Generator loss: 0.700812] [Discriminator loss: 0.693722]
Epoch: 1250 [Generator loss: 0.696290] [Discriminator loss: 0.693853]
Epoch: 1500 [Generator loss: 0.695631] [Discriminator loss: 0.694164]
Epoch: 1750 [Generator loss: 0.694394] [Discriminator loss: 0.694034]
Epoch: 2000 [Generator loss: 0.690796] [Discriminator loss: 0.694117]
Epoch: 2250 [Generator loss: 0.689114] [Discriminator loss: 0.694086]
Epoch: 2500 [Generator loss: 0.689568] [Discriminator loss: 0.693856]
Epoch: 2750 [Generator loss: 0.688964] [Discriminator loss: 0.693701]
Epoch: 3000 [Generator loss: 0.691874] [Discriminator loss: 0.693466]
Epoch: 3250 [Generator loss: 0.692807] [Discriminator loss: 0.693696]
Epoch: 3500 [Generator loss: 0.690907] [Discriminator loss: 0.693512]
Epoch: 3750 [Generator loss: 0.691412] [Discriminator loss: 0.694885]
Epoch: 4000 [Generator loss: 0.691537] [Discriminator loss: 0.693633]
Epoch: 4250 [Generator loss: 0.695284] [Discriminator loss: 0.693675]
Epoch: 4500 [Generator loss: 0.692859] [Discriminator loss: 0.694112]
Epoch: 4750 [Generator loss: 0.693220] [Discriminator loss: 0.693671]
Epoch: 5000 [Generator loss: 0.693235] [Discriminator loss: 0.693470]
```



Generating numerical data with generator

In [0]:
```
noise = np.random.normal(0, 1, (10000, latent_dim))
generated_numerical_data = generator.predict(noise)
generated_numerical_data
```

Out[0]:
```
array([[0.15321296, 0.46257776, 0.3876037 , 0.4461066 ],
       [0.3510152 , 0.4957723 , 0.40183526, 0.45575505],
       [0.52322143, 0.5801354 , 0.5952139 , 0.7163593 ],
       ...,
       [0.9084321 , 0.24215817, 0.20422834, 0.29189658],
       [0.08021015, 0.29829642, 0.22115415, 0.26953667],
       [0.64476985, 0.6214607 , 0.5681896 , 0.6198517 ]], dtype=float32)
```

Converting the generated data to similar as the original

In [0]:
```
generated_numerical_data = mms.inverse_transform(generated_numerical_data)
gen_df = pd.DataFrame(data = generated_numerical_data, columns = numerical_data.columns)
gen_df
```

Out[0]:

	patient_id	Height	Weight	bmi
0	15321.385742	1.668914	65.440277	23.292553
1	35097.851562	1.689770	66.765488	23.462173
2	52315.199219	1.742774	84.772476	28.043596
3	33206.734375	1.627241	56.365398	21.111996
4	61771.718750	1.709197	81.052765	27.745068
...	...	...	...	...
9995	14456.940430	1.631306	72.679932	27.473936
9996	3687.655029	1.747826	87.184990	28.467550
9997	90828.953125	1.530428	48.364773	20.581543
9998	8022.490723	1.565699	49.940865	20.188454
9999	64467.730469	1.768738	82.256042	26.346992

10000 rows × 4 columns

Comaparing original and generated data

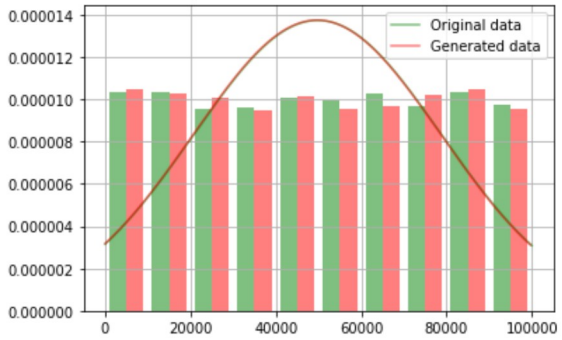
- Normal distribution
- mean, std, var
- correlation matrix

In [0]:
```
def normal_distribution(r, f):

        r_x = np.linspace(r.min(), r.max(), len(r))
        f_x = np.linspace(f.min(), f.max(), len(f))

        r_y = scipy.stats.norm.pdf(r_x, r.mean(), r.std())
        f_y = scipy.stats.norm.pdf(f_x, f.mean(), f.std())

        n, bins, patches = plt.hist([r, f], density = True, alpha = 0.5, color = ["green", "red"])
        xmin, xmax = plt.xlim()

        plt.plot(r_x, r_y, color = "green", label = "Original data", alpha = 0.5)
        plt.plot(f_x, f_y, color = "red", label = "Generated data", alpha = 0.5)
        title = f"Original data mean {np.round(r.mean(), 4)}, Original data std {np.round(r.std(), 4)}, Ori
ginal data var {np.round(r.var(), 4)}\nGenerated data mean {np.round(f.mean(), 4)}, Generated data {np.roun
d(f.std(), 4)}, Generated data var {np.round(f.var(), 2)}"
        plt.title(title)
        plt.legend()
        plt.grid()
        plt.show()
```

In [0]:
```
for column in gen_df.columns:
    print(column, "Normal distribution")
    normal_distribution(numerical_data[column], gen_df[column])
```

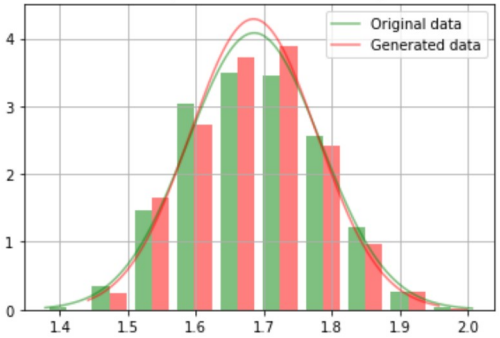
```
patient_id Normal distribution
```

Original data mean 49803.0952, Original data std 29018.0358, Original data var 842046401.9473
Generated data mean 49641.9883, Generated data 28994.8965, Generated data var 840704064.0



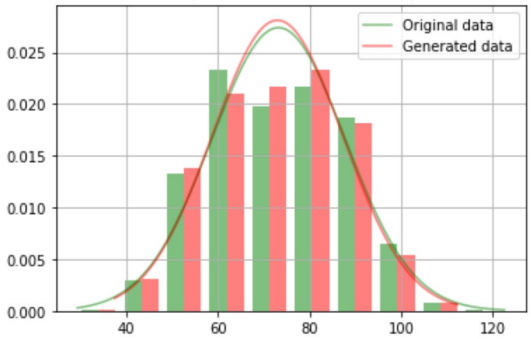

```
Height Normal distribution
```

Original data mean 1.6857, Original data std 0.0977, Original data var 0.0096
Generated data mean 1.6848, Generated data 0.0931, Generated data var 0.01



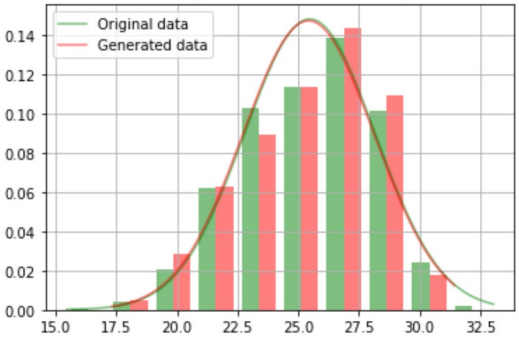

```
Weight Normal distribution
```

Original data mean 73.2281, Original data std 14.5641, Original data var 212.1143
Generated data mean 72.9955, Generated data 14.1937, Generated data var 201.46




```
bmi Normal distribution
```

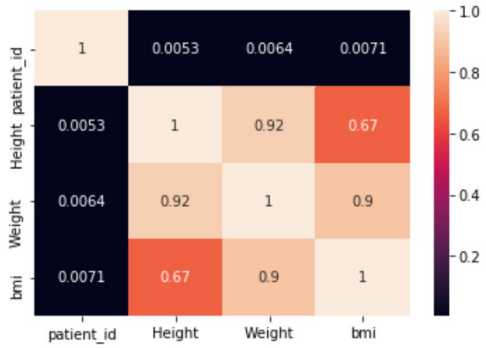
Original data mean 25.4754, Original data std 2.6938, Original data var 7.2567
Generated data mean 25.4162, Generated data 2.7081, Generated data var 7.33



In [0]:
```
#correlation matrix compareing
print("Original data")
sns.heatmap(numerical_data.corr(), annot = True)
```

```
Original data
```

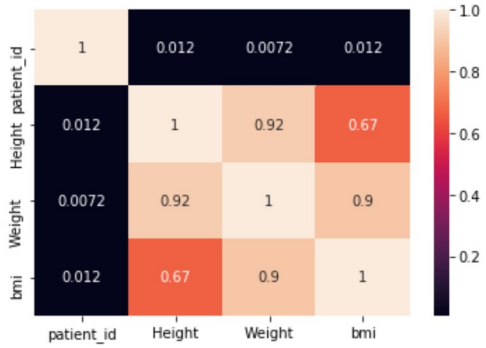
Out[0]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa3d4b0fe10>



In [0]:
```
print("Generated data")
sns.heatmap(gen_df.corr(), annot = True)
```

```
Generated data
```

Out[0]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa3d4a59fd0>



Results

- normal distribution, mean, standard deviation and variance are very similar compared to the original
- The correlation is very similar compared to the original

GAN for categorical data

- from the first example change only the generators last activation function to softmax and change discriminator input size to correspond to generator output
- Testing heterogeneous and homogeneous data

heterogeneous data

Generator

In [0]:
```
def build_generator(n_columns, latent_dim):
    model = Sequential()
    model.add(Dense(32, kernel_initializer = "he_uniform", input_dim = latent_dim))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum = 0.8))
    model.add(Dense(64,  kernel_initializer = "he_uniform"))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum= 0.8))
    model.add(Dense(128,  kernel_initializer = "he_uniform"))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum = 0.8))
    model.add(Dense(8,  kernel_initializer = "he_uniform"))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum = 0.8))
    model.add(Dense(n_columns, activation = "softmax"))
    return model
```

In [0]:
```
latent_dim = 100
generator2 = build_generator(bmi_class_ohe.shape[1], latent_dim)
```

Discriminator

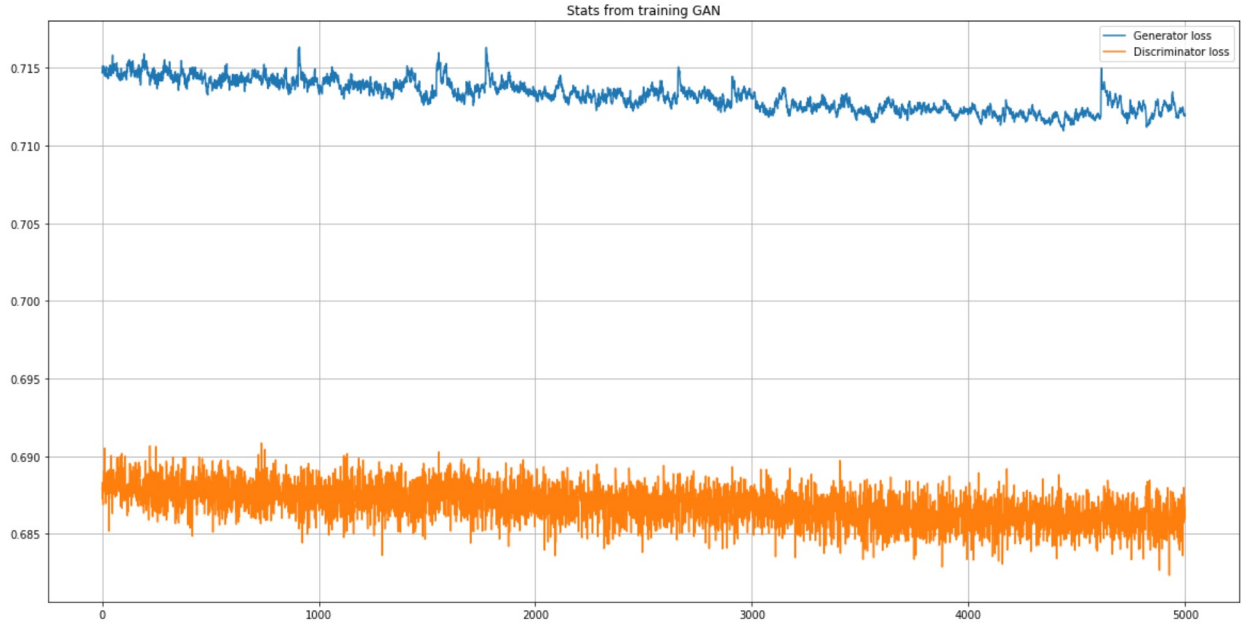
In [0]:
```
discriminator2 = build_discriminator(bmi_class_ohe.shape[1])
```

GAN and training

In [0]:
```
gan2 = build_gan(generator2, discriminator2)
```

In [0]:
```
train(gan2, generator2, discriminator2, bmi_class_ohe.values, latent_dim, n_epochs = 5000, n_batch = 8000,
n_eval = 100)
```

```
Epoch: 100 [Generator loss: 0.714700] [Discriminator loss: 0.688783]
Epoch: 200 [Generator loss: 0.715091] [Discriminator loss: 0.688285]
Epoch: 300 [Generator loss: 0.715068] [Discriminator loss: 0.687854]
Epoch: 400 [Generator loss: 0.714152] [Discriminator loss: 0.687839]
Epoch: 500 [Generator loss: 0.714033] [Discriminator loss: 0.686781]
Epoch: 600 [Generator loss: 0.714147] [Discriminator loss: 0.686962]
Epoch: 700 [Generator loss: 0.714761] [Discriminator loss: 0.687270]
Epoch: 800 [Generator loss: 0.714418] [Discriminator loss: 0.687299]
Epoch: 900 [Generator loss: 0.714241] [Discriminator loss: 0.687551]
Epoch: 1000 [Generator loss: 0.713843] [Discriminator loss: 0.686919]
Epoch: 1100 [Generator loss: 0.713712] [Discriminator loss: 0.686342]
Epoch: 1200 [Generator loss: 0.714151] [Discriminator loss: 0.685900]
Epoch: 1300 [Generator loss: 0.713388] [Discriminator loss: 0.688188]
Epoch: 1400 [Generator loss: 0.714334] [Discriminator loss: 0.688902]
Epoch: 1500 [Generator loss: 0.713345] [Discriminator loss: 0.688169]
Epoch: 1600 [Generator loss: 0.714360] [Discriminator loss: 0.687553]
Epoch: 1700 [Generator loss: 0.713862] [Discriminator loss: 0.687548]
Epoch: 1800 [Generator loss: 0.714293] [Discriminator loss: 0.687084]
Epoch: 1900 [Generator loss: 0.713734] [Discriminator loss: 0.687385]
Epoch: 2000 [Generator loss: 0.713278] [Discriminator loss: 0.685864]
Epoch: 2100 [Generator loss: 0.713681] [Discriminator loss: 0.687302]
Epoch: 2200 [Generator loss: 0.713412] [Discriminator loss: 0.685138]
Epoch: 2300 [Generator loss: 0.712654] [Discriminator loss: 0.687019]
Epoch: 2400 [Generator loss: 0.713180] [Discriminator loss: 0.686486]
Epoch: 2500 [Generator loss: 0.713635] [Discriminator loss: 0.686041]
Epoch: 2600 [Generator loss: 0.712787] [Discriminator loss: 0.687344]
Epoch: 2700 [Generator loss: 0.713299] [Discriminator loss: 0.687123]
Epoch: 2800 [Generator loss: 0.713195] [Discriminator loss: 0.686152]
Epoch: 2900 [Generator loss: 0.712537] [Discriminator loss: 0.687371]
Epoch: 3000 [Generator loss: 0.713039] [Discriminator loss: 0.687127]
Epoch: 3100 [Generator loss: 0.712100] [Discriminator loss: 0.687029]
Epoch: 3200 [Generator loss: 0.712599] [Discriminator loss: 0.686374]
Epoch: 3300 [Generator loss: 0.712335] [Discriminator loss: 0.685326]
Epoch: 3400 [Generator loss: 0.712223] [Discriminator loss: 0.684463]
Epoch: 3500 [Generator loss: 0.712305] [Discriminator loss: 0.687696]
Epoch: 3600 [Generator loss: 0.712382] [Discriminator loss: 0.685891]
Epoch: 3700 [Generator loss: 0.712267] [Discriminator loss: 0.686724]
Epoch: 3800 [Generator loss: 0.712431] [Discriminator loss: 0.686962]
Epoch: 3900 [Generator loss: 0.712537] [Discriminator loss: 0.686883]
Epoch: 4000 [Generator loss: 0.712354] [Discriminator loss: 0.685720]
Epoch: 4100 [Generator loss: 0.712537] [Discriminator loss: 0.685495]
Epoch: 4200 [Generator loss: 0.712001] [Discriminator loss: 0.686904]
Epoch: 4300 [Generator loss: 0.712313] [Discriminator loss: 0.686851]
Epoch: 4400 [Generator loss: 0.712032] [Discriminator loss: 0.685922]
Epoch: 4500 [Generator loss: 0.711955] [Discriminator loss: 0.686325]
Epoch: 4600 [Generator loss: 0.712068] [Discriminator loss: 0.685680]
Epoch: 4700 [Generator loss: 0.713238] [Discriminator loss: 0.686022]
Epoch: 4800 [Generator loss: 0.712319] [Discriminator loss: 0.686051]
Epoch: 4900 [Generator loss: 0.712626] [Discriminator loss: 0.685036]
Epoch: 5000 [Generator loss: 0.712034] [Discriminator loss: 0.686676]
```



Generating heterogeneous categorical data with generator

In [0]:
```
noise = np.random.normal(0, 1, (10000, latent_dim))
generated_categorical_data = generator2.predict(noise)
generated_categorical_data
```

Out[0]:
```
array([[7.4237496e-06, 9.9998379e-01, 6.6982466e-06, 2.1381011e-06],
       [4.0223238e-08, 9.9999905e-01, 6.2216714e-07, 3.7322494e-07],
       [1.0000000e+00, 9.0978386e-18, 3.0524983e-09, 3.7104322e-10],
       ...,
       [1.1083174e-10, 1.0000000e+00, 8.3294704e-09, 1.1884604e-08],
       [3.2747508e-08, 9.9999893e-01, 6.3820426e-07, 4.5642687e-07],
       [5.8153881e-07, 9.9999774e-01, 8.7461643e-07, 8.4739696e-07]],
      dtype=float32)
```

Generated data to same form as original data

In [0]:
```
gen_df2 = pd.DataFrame(data = np.round(generated_categorical_data), columns = bmi_class_ohe.columns) #round ohe values
gen_df2
```

Out[0]:

	bmi_class_healthy	bmi_class_overweight	bmi_class_severely overweight	bmi_class_underweight
0	0.0	1.0	0.0	0.0
1	0.0	1.0	0.0	0.0
2	1.0	0.0	0.0	0.0
3	0.0	1.0	0.0	0.0
4	0.0	1.0	0.0	0.0
...	...	...	...	...
9995	0.0	1.0	0.0	0.0
9996	1.0	0.0	0.0	0.0
9997	0.0	1.0	0.0	0.0
9998	0.0	1.0	0.0	0.0
9999	0.0	1.0	0.0	0.0

10000 rows × 4 columns

Comapring generated and original data

- Count generated and original ohe values

In [0]:
```
#original data value coint
for column in gen_df2.columns:
    print(bmi_class_ohe[column].value_counts())
```

```
0    5888
1    4112
Name: bmi_class_healthy, dtype: int64
1    5590
0    4410
Name: bmi_class_overweight, dtype: int64
0    9744
1     256
Name: bmi_class_severely overweight, dtype: int64
0    9958
1      42
Name: bmi_class_underweight, dtype: int64
```

In [0]:
```
#generated data value count
for column in gen_df2.columns:
    print(gen_df2[column].value_counts())
```

```
0.0    5756
1.0    4244
Name: bmi_class_healthy, dtype: int64
1.0    5756
0.0    4244
Name: bmi_class_overweight, dtype: int64
0.0    10000
Name: bmi_class_severely overweight, dtype: int64
0.0    10000
Name: bmi_class_underweight, dtype: int64
```

homogeneous data

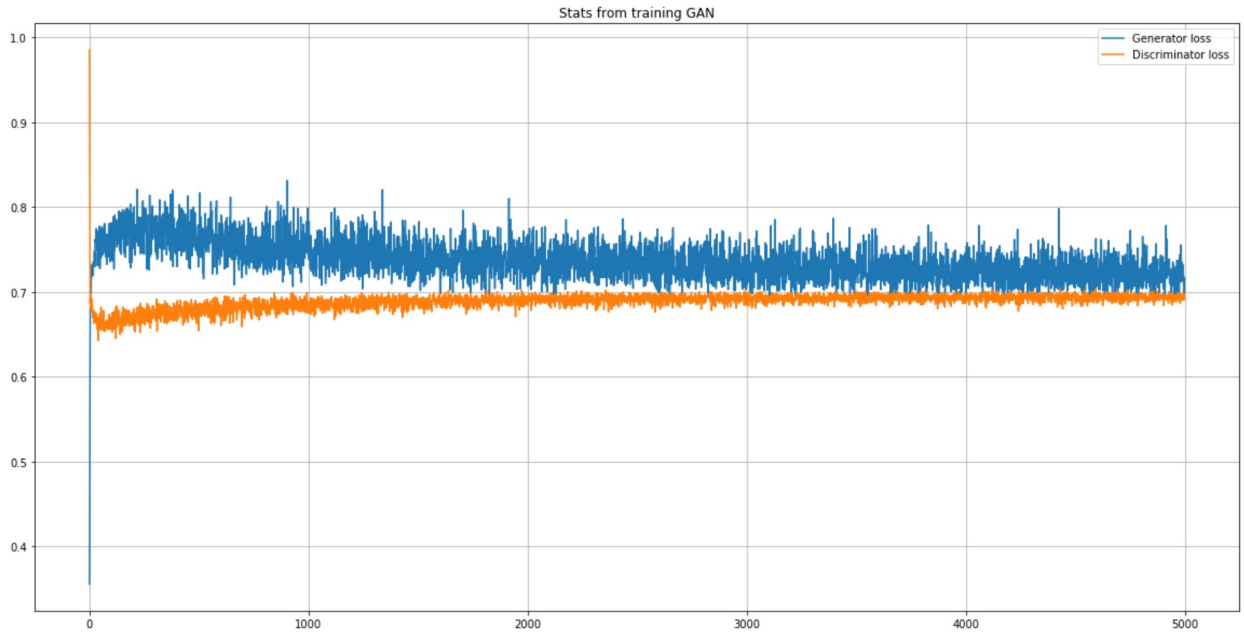
In [0]:
```
generator3 = build_generator(gender_ohe.shape[1], 100) #Generator
```

In [0]:
```
discriminator3 = build_discriminator(gender_ohe.shape[1]) #Discriminator
```

In [0]:
```
gan3 = build_gan(generator3, discriminator3) #GAN
```

In [0]:
```
train(gan3, generator3, discriminator3, gender_ohe.values, latent_dim, n_epochs = 5000, n_batch = 256, n_eval = 250) #GAN training
```

```
Epoch: 250 [Generator loss: 0.757127] [Discriminator loss: 0.676114]
Epoch: 500 [Generator loss: 0.749862] [Discriminator loss: 0.653795]
Epoch: 750 [Generator loss: 0.729575] [Discriminator loss: 0.681998]
Epoch: 1000 [Generator loss: 0.742853] [Discriminator loss: 0.685000]
Epoch: 1250 [Generator loss: 0.756058] [Discriminator loss: 0.679322]
Epoch: 1500 [Generator loss: 0.729207] [Discriminator loss: 0.685211]
Epoch: 1750 [Generator loss: 0.715714] [Discriminator loss: 0.681153]
Epoch: 2000 [Generator loss: 0.719036] [Discriminator loss: 0.693484]
Epoch: 2250 [Generator loss: 0.742653] [Discriminator loss: 0.685905]
Epoch: 2500 [Generator loss: 0.726150] [Discriminator loss: 0.688196]
Epoch: 2750 [Generator loss: 0.713473] [Discriminator loss: 0.688078]
Epoch: 3000 [Generator loss: 0.705491] [Discriminator loss: 0.690643]
Epoch: 3250 [Generator loss: 0.728701] [Discriminator loss: 0.686508]
Epoch: 3500 [Generator loss: 0.730668] [Discriminator loss: 0.696081]
Epoch: 3750 [Generator loss: 0.698900] [Discriminator loss: 0.688219]
Epoch: 4000 [Generator loss: 0.713623] [Discriminator loss: 0.686297]
Epoch: 4250 [Generator loss: 0.712428] [Discriminator loss: 0.688368]
Epoch: 4500 [Generator loss: 0.717807] [Discriminator loss: 0.700603]
Epoch: 4750 [Generator loss: 0.731628] [Discriminator loss: 0.694177]
Epoch: 5000 [Generator loss: 0.713826] [Discriminator loss: 0.691747]
```



Generating homogeneous categorical data with generator

In [0]:
```
noise = np.random.normal(0, 1, (10000, 100))
generated_categorical_data2 = generator3.predict(noise)
generated_categorical_data2
```

Out[0]:
```
array([[1.0000000e+00, 8.0513279e-10],
       [9.9999332e-01, 6.6958833e-06],
       [1.4125016e-10, 1.0000000e+00],
       ...,
       [2.1017400e-11, 1.0000000e+00],
       [1.0000000e+00, 1.5108084e-12],
       [1.0000000e+00, 3.3435496e-19]], dtype=float32)
```

Generated data to same form as original data

In [0]:
```
gen_df3 = pd.DataFrame(data = np.round(generated_categorical_data2), columns = gender_ohe.columns)
gen_df3
```

Out[0]:

	Gender_Female	Gender_Male
0	1.0	0.0
1	1.0	0.0
2	0.0	1.0
3	0.0	1.0
4	0.0	1.0
...	...	...
9995	0.0	1.0
9996	1.0	0.0
9997	0.0	1.0
9998	1.0	0.0
9999	1.0	0.0

10000 rows × 2 columns

Comapring generated and original data

- Count generated and original ohe values

In [0]:
```
#original data value count
for column in gen_df3.columns:
    print(gender_ohe[column].value_counts())
```

```
1    5000
0    5000
Name: Gender_Female, dtype: int64
1    5000
0    5000
Name: Gender_Male, dtype: int64
```

In [0]:
```
#generated data value count
for column in gen_df3.columns:
    print(gen_df3[column].value_counts())
```

```
0.0    5018
1.0    4982
Name: Gender_Female, dtype: int64
1.0    5018
0.0    4982
Name: Gender_Male, dtype: int64
```

Results

- A heterogeneous data will cause problems if the amount of some variable is too small for the size of the dataset. The generator does not generate this small amount of data at all (bmi_class_underweight 42pcs out of 10000)
- Generated homogenous data is very similar compared to the original (under 1% error of generating same amount of categorical ohe varaibles)

Generating categorical and numerical data with GAN

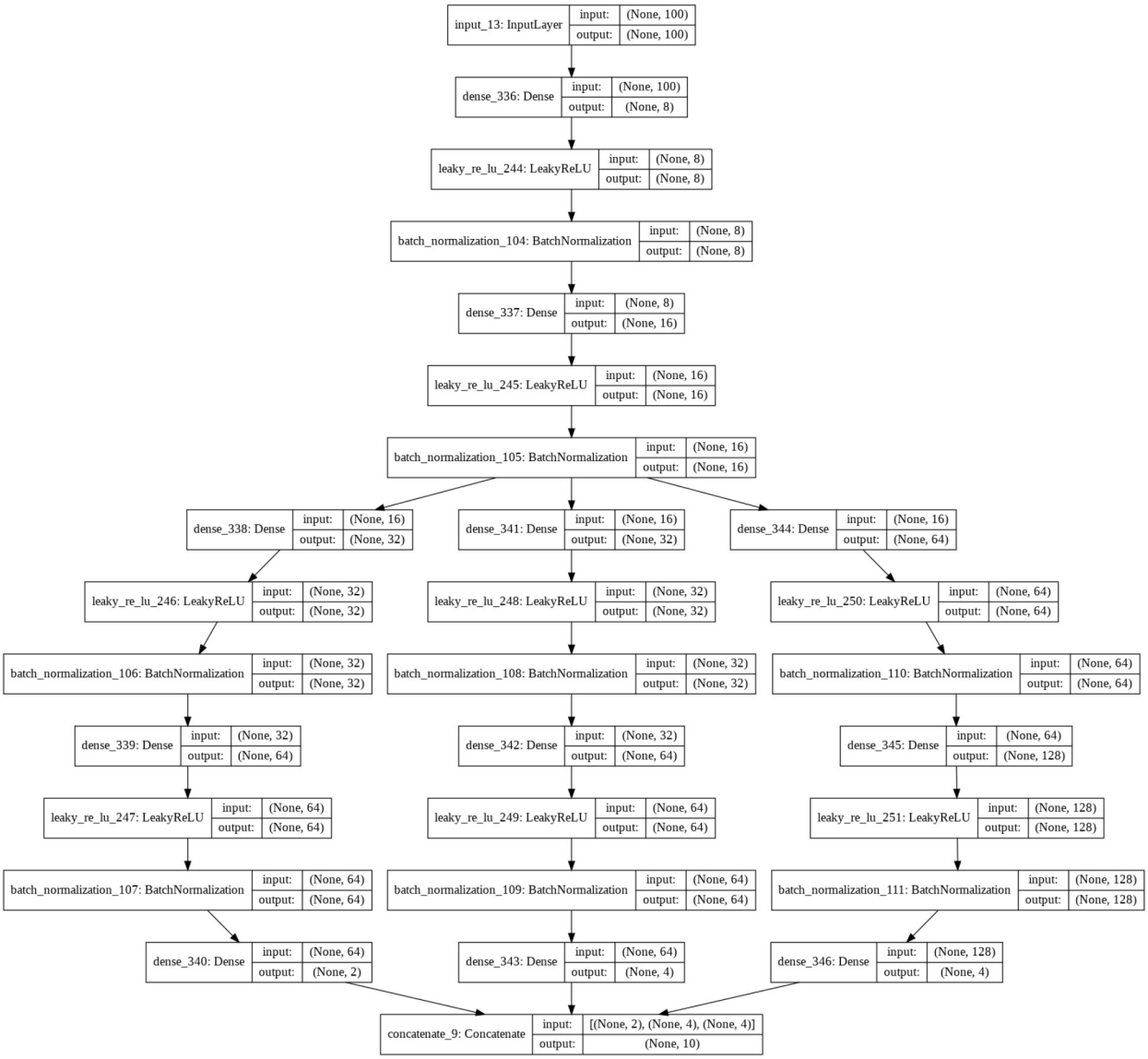
Generator

- Generator takes input from the random normal distribution (latent_dim)
- Generator handles numerical and ohe values in different branches
- Combine different branch outputs as one output (generator output)

In [0]:
```
def build_generator(categorical_data_shape, categorical_data_shape2, numerical_data_shape):
    #noise as input from the latent space
    noise = Input(shape = (100,))
    hidden_1 = Dense(8, kernel_initializer = "he_uniform")(noise)
    hidden_1 = LeakyReLU(0.2)(hidden_1)
    hidden_1 = BatchNormalization(momentum = 0.8)(hidden_1)

    hidden_2 = Dense(16, kernel_initializer = "he_uniform")(hidden_1)
    hidden_2 = LeakyReLU(0.2)(hidden_2)
    hidden_2 = BatchNormalization(momentum = 0.8)(hidden_2)

    #Branch 1 for generating categorical gender data
    branch_1 = Dense(32, kernel_initializer = "he_uniform")(hidden_2)
    branch_1 = LeakyReLU(0.2)(branch_1)
    branch_1 = BatchNormalization(momentum = 0.8)(branch_1)

    branch_1 = Dense(64, kernel_initializer = "he_uniform")(branch_1)
    branch_1 = LeakyReLU(0.2)(branch_1)
    branch_1 = BatchNormalization(momentum=0.8)(branch_1)
    #Output 1 layer, softmax activation for multi classification
    branch_1_output = Dense(categorical_data_shape, activation = "softmax")(branch_1)

    #Branch 2 for generating categorical bmi_class data
    branch_2 = Dense(32, kernel_initializer = "he_uniform")(hidden_2)
    branch_2 = LeakyReLU(0.2)(branch_2)
    branch_2 = BatchNormalization(momentum=0.8)(branch_2)

    branch_2 = Dense(64, kernel_initializer = "he_uniform")(branch_2)
    branch_2 = LeakyReLU(0.2)(branch_2)
    branch_2 = BatchNormalization(momentum=0.8)(branch_2)
    #Output 2 layer, softmax activation for multi classification
    branch_2_output = Dense(categorical_data_shape2, activation = "softmax")(branch_2)

    #Branch 3 for generating numerical data
    branch_3 = Dense(64, kernel_initializer = "he_uniform")(hidden_2)
    branch_3 = LeakyReLU(0.2)(branch_3)
    branch_3 = BatchNormalization(momentum=0.8)(branch_3)

    branch_3 = Dense(128, kernel_initializer = "he_uniform")(branch_3)
    branch_3 = LeakyReLU(0.2)(branch_3)
    branch_3 = BatchNormalization(momentum=0.8)(branch_3)
    #Output 3, sigmoid activation
    branch_3_output = Dense(numerical_data_shape, activation = "sigmoid")(branch_3_hidden_2)

    #Combined output
    combined_output = concatenate([branch_1_output, branch_2_output, branch_3_output])
    #Return model
    return Model(inputs = noise, outputs = combined_output)
```

In [0]:
```
generator4 = build_generator(gender_ohe.shape[1], bmi_class_ohe.shape[1], numerical_data_rescaled.shape[1])
plot_model(generator4, show_layer_names = True, show_shapes = True)
```

Out[0]:



Discriminator

- Input is size of the data
- Outputs classification

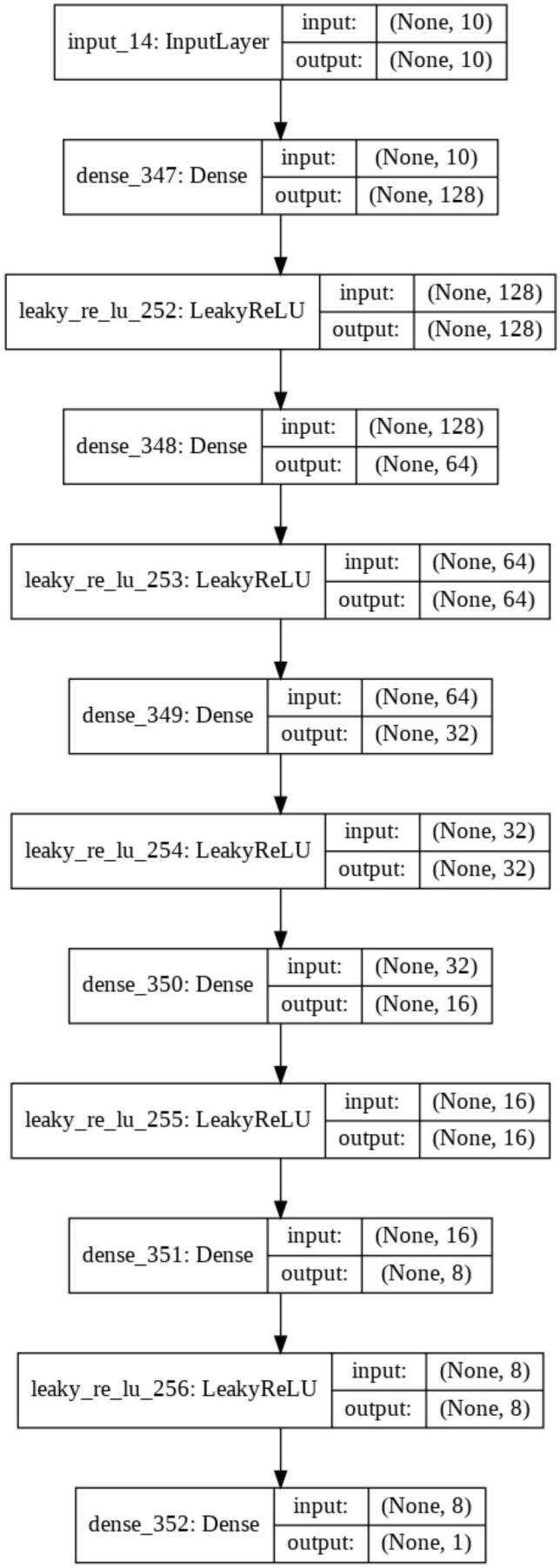
In [0]:
```
def build_discriminator(inputs_n):
    #Input from generator
    d_input = Input(shape = (inputs_n,))
    d = Dense(128, kernel_initializer="he_uniform")(d_input)
    d = LeakyReLU(0.2)(d)
    d = Dense(64, kernel_initializer="he_uniform")(d)
    d = LeakyReLU(0.2)(d)
    d = Dense(32, kernel_initializer="he_uniform")(d)
    d = LeakyReLU(0.2)(d)
    d = Dense(16, kernel_initializer="he_uniform")(d)
    d = LeakyReLU(0.2)(d)
    d = Dense(8, kernel_initializer="he_uniform")(d)
    d = LeakyReLU(0.2)(d)
    #Discriminator output for classification, sigmoid activation
    d_output = Dense(1, activation = "sigmoid")(d)
    #compile and return model
    model = Model(inputs = d_input, outputs = d_output)
    model.compile(loss = "binary_crossentropy", optimizer = optimizer, metrics = ["accuracy"])
    return model
```

In [0]:
```
inputs_n = gender_ohe.shape[1] + bmi_class_ohe.shape[1] + numerical_data_rescaled.shape[1]
inputs_n
```

Out[0]: 10

In [0]:
```
discriminator4 = build_discriminator(inputs_n)
plot_model(discriminator4, show_layer_names = True, show_shapes = True)
```

Out[0]:



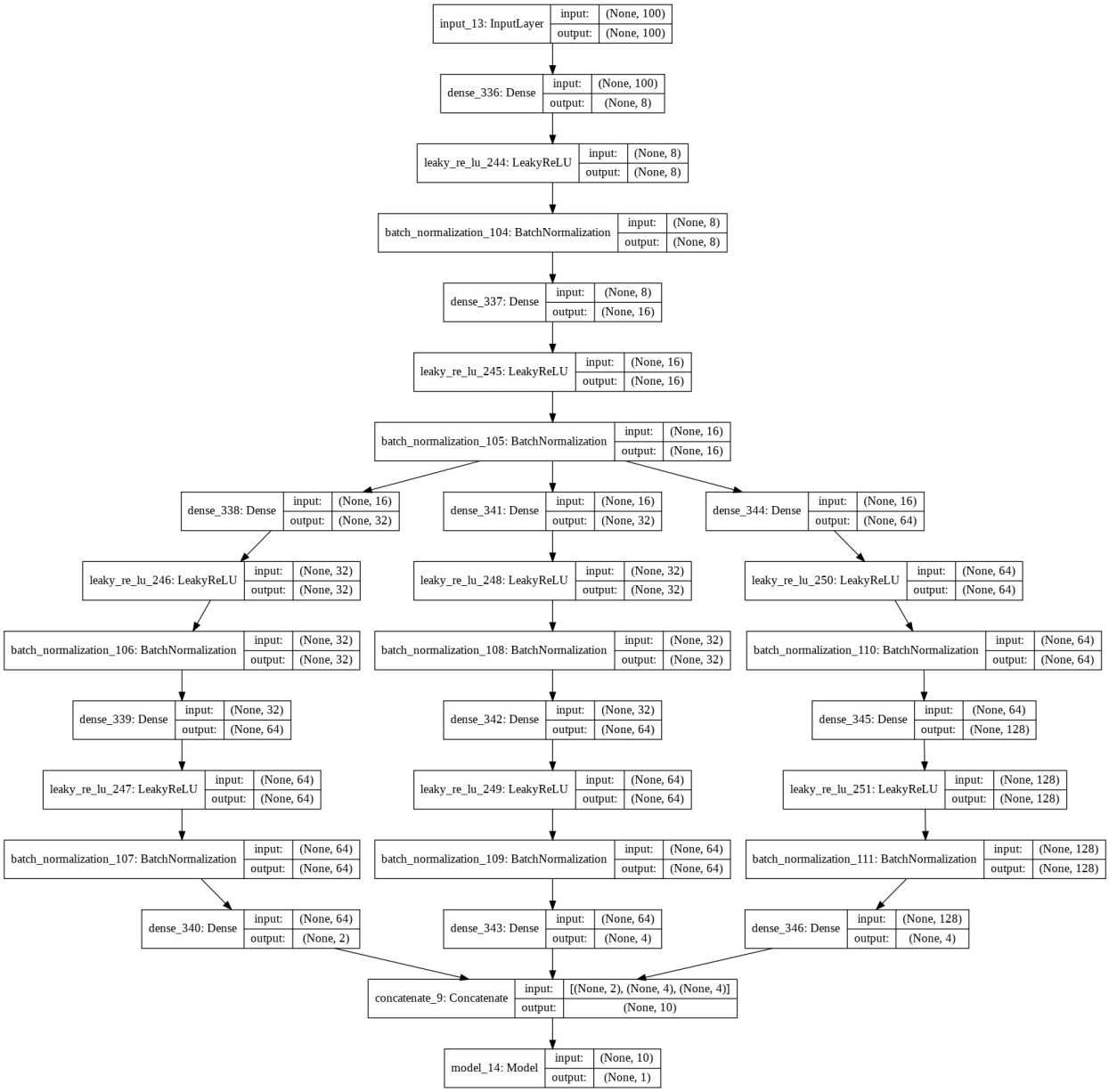
GAN

- GAN input is generator input
- GAN output is discriminator output

In [0]:
```
def build_gan(generator, discriminator):
    #Make discriminator not trainable
    discriminator.trainable = False
    #Discriminator takes input from generator and make discriminator GAN output
    gan_output = discriminator(generator.output)
    #Initialize gan
    model = Model(inputs = generator.input, outputs = gan_output)
    #Compile model
    model.compile(loss = "binary_crossentropy", optimizer = optimizer)
    #Return Model
    return model
```

In [0]:
```
gan4 = build_gan(generator4, discriminator4)
plot_model(gan4, show_layer_names = True, show_shapes = True)
```

Out[0]:

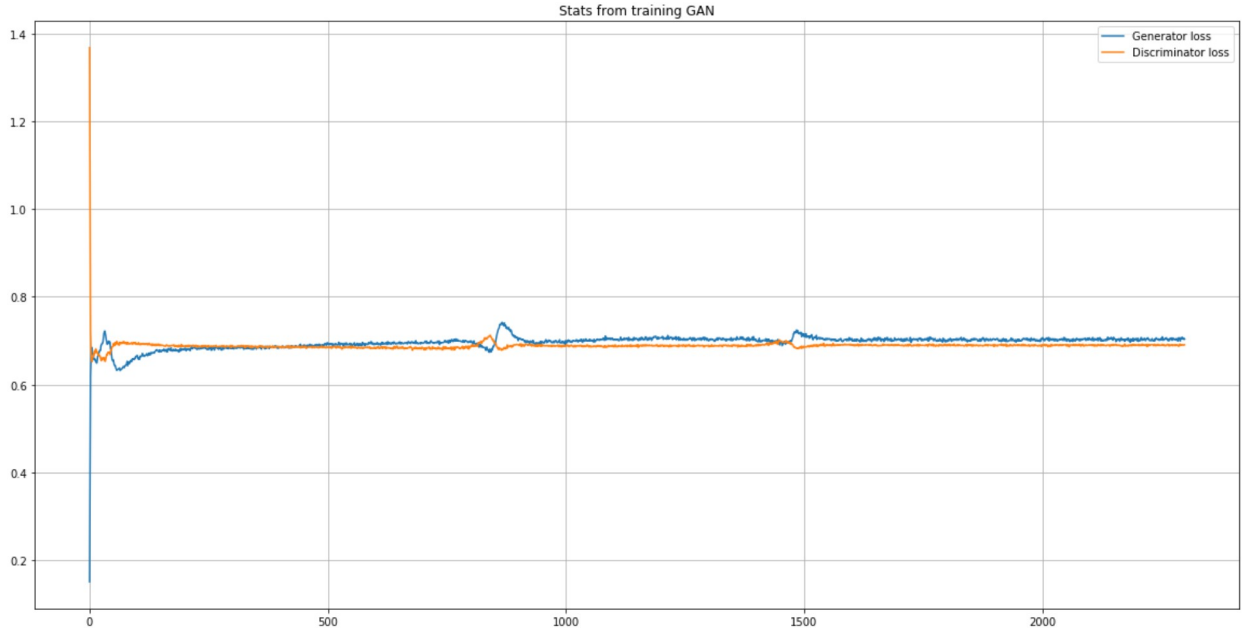


GAN training

In [0]:
```
def train(gan, generator, discriminator, categorical_data, categorical_data2, numerical_data, latent_dim, n
_epochs, n_batch, n_eval):
    #Half batch size for updateting discriminator
    half_batch = int(n_batch / 2)

    #lists for stats from the model
    discriminator_loss = []
    generator_loss = []

    #generate class labels for fake and real
    valid = np.ones((half_batch, 1))
    y_gan = np.ones((n_batch, 1))
    fake = np.zeros((half_batch, 1))
    #training loop
    for i in range(n_epochs):

        #select random batch from real categorical and numerical data
        idx = np.random.randint(0, categorical_data.shape[0], half_batch)
        gender_real = categorical_data[idx]
        bmi_real = categorical_data2[idx]
        numerical_real = numerical_data[idx]

        #concatenate categorical and numerical data for the discriminator
        real_data = np.concatenate([gender_real, bmi_real, numerical_real], axis = 1)

        #generate fake samples from the noise
        noise = np.random.normal(0, 1, (half_batch, latent_dim))
        fake_data = generator.predict(noise)

        #train the discriminator and return losses and acc
        d_loss_real, da_real = discriminator.train_on_batch(real_data, valid)
        d_loss_fake, da_fake = discriminator.train_on_batch(fake_data, fake)

        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
        discriminator_loss.append(d_loss)

        #generate noise for generator input and train the generator (to have the discriminator label sample
s as valid)
        noise = np.random.normal(0, 1, (n_batch, latent_dim))
        g_loss = gan.train_on_batch(noise, y_gan)

        generator_loss.append(g_loss)
        #evaluate progress
        if (i+1) % n_eval == 0:
            print ("Epoch: %d [Discriminator loss: %f] [Generator loss: %f]" % (i + 1, d_loss, g_loss))
    plt.figure(figsize = (20, 10))
    plt.plot(generator_loss, label = "Generator loss")
    plt.plot(discriminator_loss, label = "Discriminator loss")
    plt.title("Stats from training GAN")
    plt.grid()
    plt.legend()
```

In [0]:
```
latent_dim = 100
train(gan4, generator4, discriminator4, gender_ohe.values, bmi_class_ohe.values, numerical_data_rescaled, l
atent_dim, n_epochs = 2300, n_batch = 2048, n_eval = 200)
```

```
Epoch: 200 [Discriminator loss: 0.687793] [Generator loss: 0.679744]
Epoch: 400 [Discriminator loss: 0.686592] [Generator loss: 0.686811]
Epoch: 600 [Discriminator loss: 0.682664] [Generator loss: 0.692192]
Epoch: 800 [Discriminator loss: 0.686256] [Generator loss: 0.696813]
Epoch: 1000 [Discriminator loss: 0.687463] [Generator loss: 0.700255]
Epoch: 1200 [Discriminator loss: 0.690072] [Generator loss: 0.706654]
Epoch: 1400 [Discriminator loss: 0.688019] [Generator loss: 0.703223]
Epoch: 1600 [Discriminator loss: 0.691656] [Generator loss: 0.701813]
Epoch: 1800 [Discriminator loss: 0.690267] [Generator loss: 0.700814]
Epoch: 2000 [Discriminator loss: 0.690937] [Generator loss: 0.706095]
Epoch: 2200 [Discriminator loss: 0.690022] [Generator loss: 0.700005]
```



Generating categorical and numerical data with generator

In [0]:
```
noise = np.random.normal(0, 1, (10000, 100))
generated_mixed_data = generator4.predict(noise)
generated_mixed_data
```

Out[0]:
```
array([[6.5177805e-09, 1.0000000e+00, 6.0876209e-01, ..., 6.1912340e-01,
        4.9177581e-01, 4.8557207e-01],
       [1.0913557e-24, 1.0000000e+00, 2.1655799e-18, ..., 5.6651312e-01,
        5.7311296e-01, 6.5712827e-01],
       [9.9999976e-01, 2.4986926e-07, 9.9999988e-01, ..., 3.1343400e-01,
        2.6918650e-01, 3.7169328e-01],
       ...,
       [1.0000000e+00, 8.3462080e-22, 1.0000000e+00, ..., 4.2388344e-01,
        3.3199945e-01, 4.4132009e-01],
       [1.1695668e-26, 1.0000000e+00, 3.8878457e-18, ..., 5.2725494e-01,
        5.7334667e-01, 7.0970201e-01],
       [2.0165229e-10, 1.0000000e+00, 3.3639376e-03, ..., 5.7177657e-01,
        4.4709170e-01, 4.0972564e-01]], dtype=float32)
```

Generated data to same form as original data

In [0]:
```
columns = list(gender_ohe.columns) + list(bmi_class_ohe.columns) + list(numerical_data.columns)
mixed_gen_df = pd.DataFrame(data = generated_mixed_data, columns = columns)
mixed_gen_df
```

Out[0]:

	Gender_Female	Gender_Male	bmi_class_healthy	bmi_class_overweight	bmi_class_severely overweight	bmi_class_underweight	patient_id	Height
0	6.517781e-09	1.000000e+00	6.087621e-01	3.890011e-01	9.748723e-04	1.261859e-03	0.071617	0.619123
1	1.091356e-24	1.000000e+00	2.165580e-18	1.000000e+00	6.474741e-11	4.974434e-10	0.307223	0.566513
2	9.999998e-01	2.498693e-07	9.999999e-01	1.527332e-16	1.202344e-07	3.369300e-10	0.103941	0.313434
3	6.787892e-18	1.000000e+00	2.128410e-22	1.000000e+00	1.516234e-11	3.028638e-10	0.339649	0.560647
4	1.000000e+00	2.398346e-21	1.000000e+00	1.086912e-20	2.371742e-11	6.409379e-13	0.479014	0.392338
...	...	...	...	...	...	...	...	...
9995	2.125468e-26	1.000000e+00	1.175566e-16	9.999919e-01	8.116408e-06	1.372956e-08	0.453946	0.588410
9996	4.573263e-27	1.000000e+00	7.991927e-19	1.000000e+00	6.310014e-12	7.932580e-11	0.535188	0.537637
9997	1.000000e+00	8.346208e-22	1.000000e+00	4.392768e-26	2.227189e-13	2.262947e-14	0.975042	0.423883
9998	1.169567e-26	1.000000e+00	3.887846e-18	1.000000e+00	1.949073e-09	2.635000e-08	0.888659	0.527255
9999	2.016523e-10	1.000000e+00	3.363938e-03	9.966144e-01	4.722630e-06	1.692283e-05	0.235413	0.571777

10000 rows × 10 columns

In [0]:
```
mixed_gen_df.iloc[:, 0:6] = np.round(mixed_gen_df.iloc[:, 0:6])
mixed_gen_df.iloc[:, 6:10] = mms.inverse_transform(mixed_gen_df.iloc[:, 6:10])
mixed_gen_df
```

Out[0]:

	Gender_Female	Gender_Male	bmi_class_healthy	bmi_class_overweight	bmi_class_severely overweight	bmi_class_underweight	patient_id	Heigl
0	0.0	1.0	1.0	0.0	0.0	0.0	7163.304199	1.76726
1	0.0	1.0	0.0	1.0	0.0	0.0	30719.494141	1.73421
2	1.0	0.0	1.0	0.0	0.0	0.0	10395.106445	1.57520
3	0.0	1.0	0.0	1.0	0.0	0.0	33961.496094	1.73052
4	1.0	0.0	1.0	0.0	0.0	0.0	47895.281250	1.62478
...	...	...	...	...	...	...	...	
9995	0.0	1.0	0.0	1.0	0.0	0.0	45388.976562	1.74797
9996	0.0	1.0	0.0	1.0	0.0	0.0	53511.605469	1.71607
9997	1.0	0.0	1.0	0.0	0.0	0.0	97488.703125	1.64460
9998	0.0	1.0	0.0	1.0	0.0	0.0	88852.062500	1.70955
9999	0.0	1.0	0.0	1.0	0.0	0.0	23539.851562	1.73752

10000 rows × 10 columns

In [0]:
```
#original data
og_data = pd.concat([gender_ohe, bmi_class_ohe, numerical_data], axis = 1)
og_data
```

Out[0]:

	Gender_Female	Gender_Male	bmi_class_healthy	bmi_class_overweight	bmi_class_severely overweight	bmi_class_underweight	patient_id	Height
0	0	1	0	0	1	0	90329	1.875714
1	0	1	1	0	0	0	82793	1.747060
2	0	1	0	1	0	0	98691	1.882397
3	0	1	0	0	1	0	20430	1.821967
4	0	1	0	1	0	0	96554	1.774998
...	...	...	...	...	...	...	...	...
9995	1	0	1	0	0	0	51506	1.680785
9996	1	0	0	1	0	0	38900	1.703506
9997	1	0	1	0	0	0	26718	1.622247
9998	1	0	1	0	0	0	67447	1.753470
9999	1	0	1	0	0	0	70063	1.573384

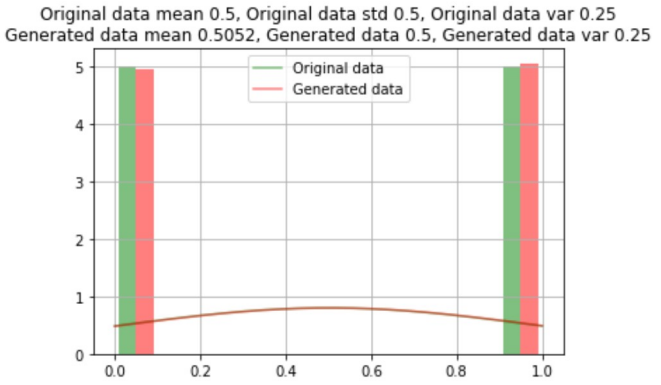
10000 rows × 10 columns

Comparing original and generated data

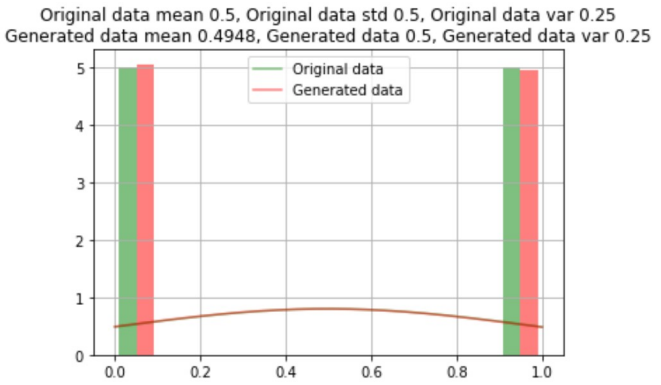
- normal distribution, mean, std, var
- correlation matrix
- counting categorical ohe variables
- neural network classifier

In [0]:
```
for column in mixed_gen_df.columns:
    print(column, "Normal distribution")
    normal_distribution(og_data[column], mixed_gen_df[column])
```

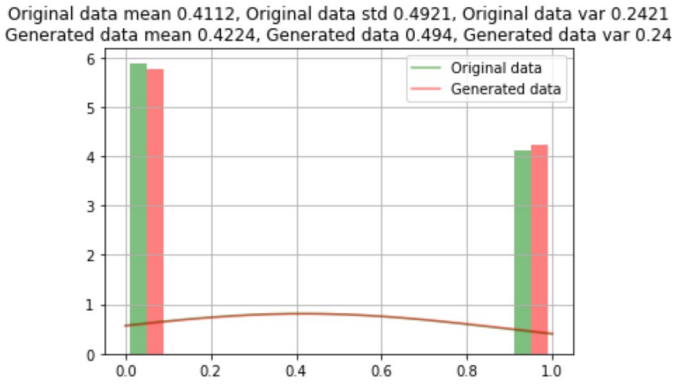
Gender_Female Normal distribution



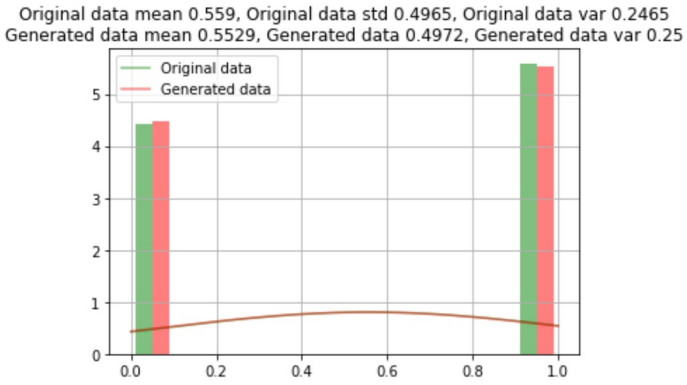
Gender_Male Normal distribution



bmi_class_healthy Normal distribution

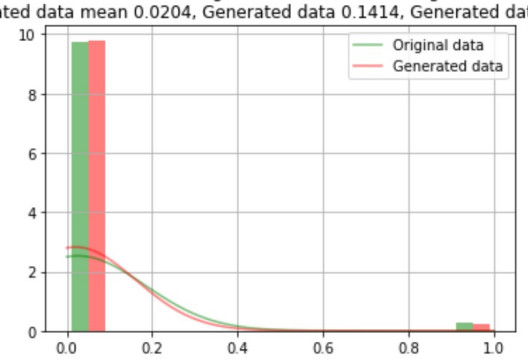


bmi_class_overweight Normal distribution



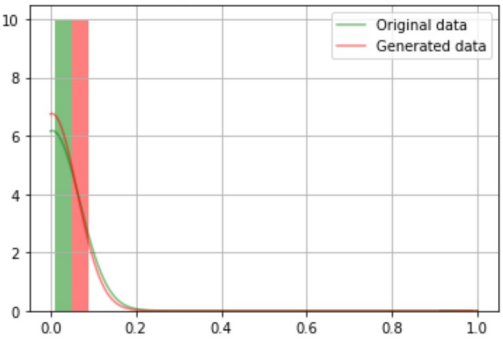
bmi_class_severely overweight Normal distribution

Original data mean 0.0256, Original data std 0.1579, Original data var 0.0249
Generated data mean 0.0204, Generated data 0.1414, Generated data var 0.02



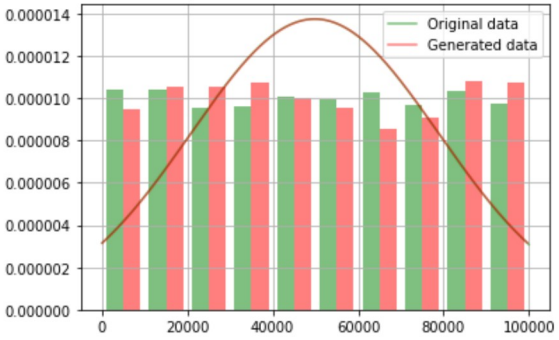
bmi_class_underweight Normal distribution

Original data mean 0.0042, Original data std 0.0647, Original data var 0.0042
Generated data mean 0.0035, Generated data 0.0591, Generated data var 0.0



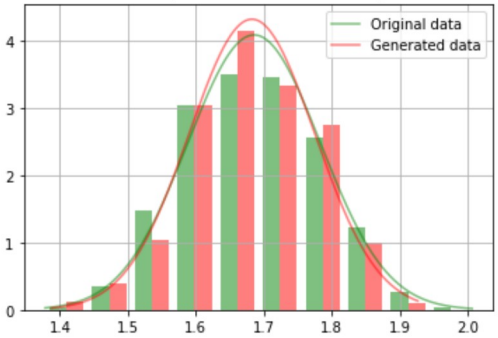
patient_id Normal distribution

Original data mean 49803.0952, Original data std 29018.0358, Original data var 842046401.9473
Generated data mean 49891.2461, Generated data 29022.9355, Generated data var 842330752.0



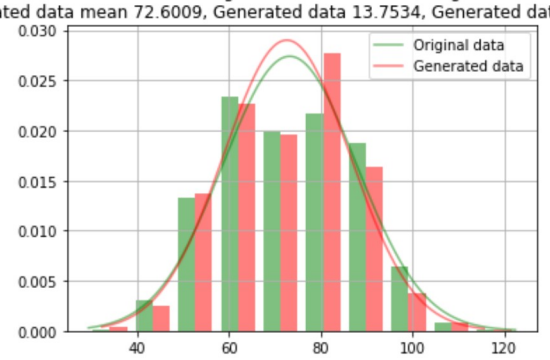
Height Normal distribution

Original data mean 1.6857, Original data std 0.0977, Original data var 0.0096
Generated data mean 1.6826, Generated data 0.0924, Generated data var 0.01



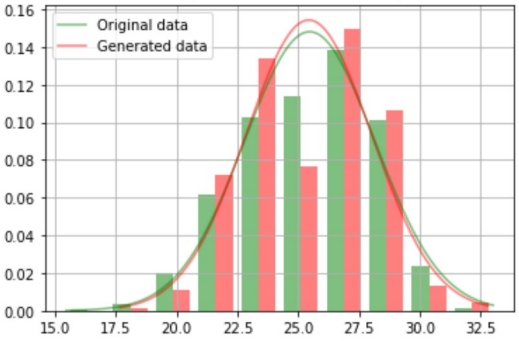
Weight Normal distribution

Original data mean 73.2281, Original data std 14.5641, Original data var 212.1143
Generated data mean 72.6009, Generated data 13.7534, Generated data var 189.16




```
bmi Normal distribution
```

Original data mean 25.4754, Original data std 2.6938, Original data var 7.2567
Generated data mean 25.441, Generated data 2.5856, Generated data var 6.69

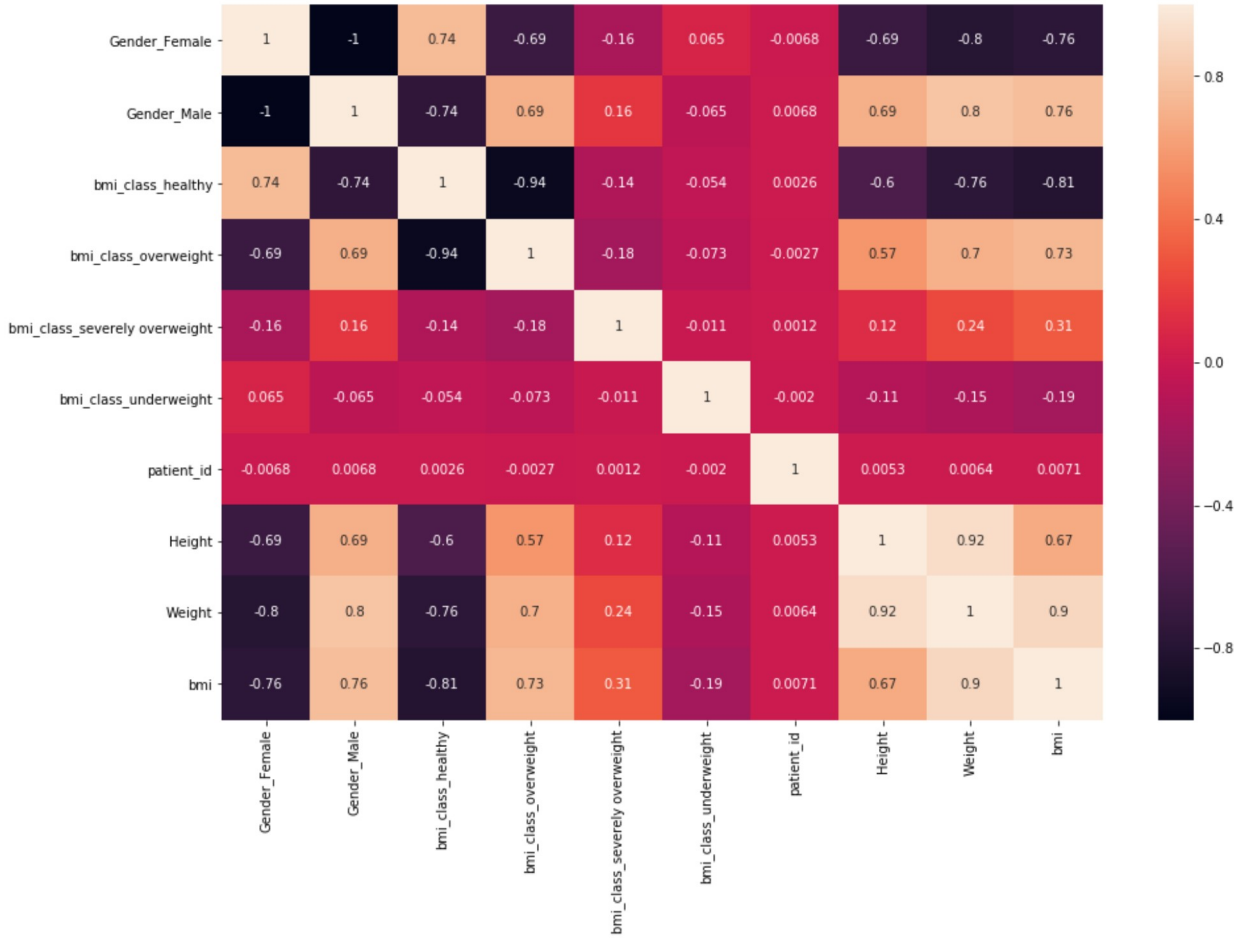


correlation matrix

In [0]:
```
#correlation matrix compareing
print("Original data")
plt.figure(figsize = (15, 10))
sns.heatmap(og_data.corr(), annot = True)
```

```
Original data
```

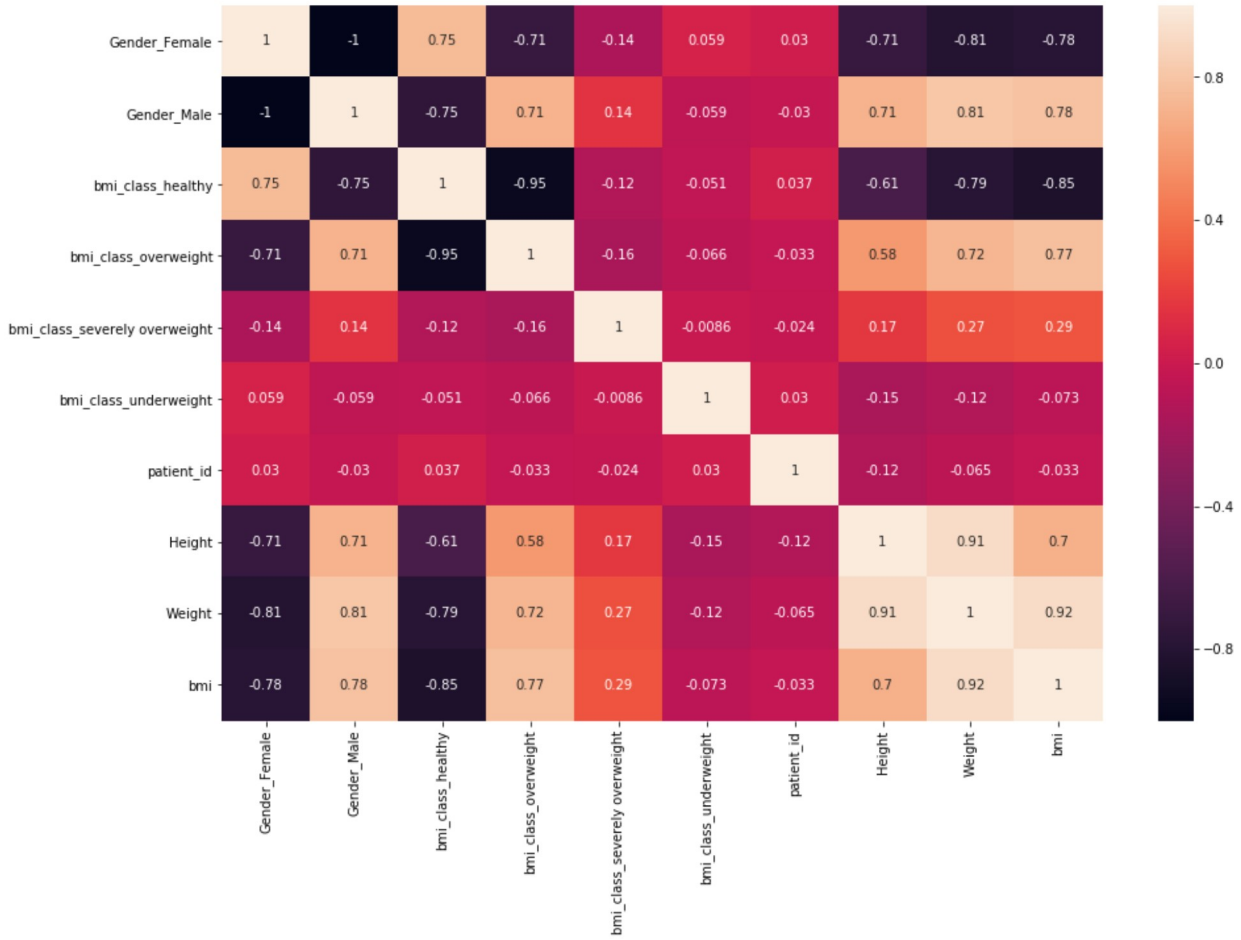
Out[0]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa3c956c438>



In [0]:
```
print("Generated data")
plt.figure(figsize = (15, 10))
sns.heatmap(mixed_gen_df.corr(), annot = True)
```

```
Generated data
```

Out[0]: <matplotlib.figure.Figure at 0x7fa3c653a7f0>



counting categorical variables

In [0]:
```
#original data value count
for column in og_data.iloc[:, 0:6].columns:
    print(og_data[column].value_counts())
```

```
1    5000
0    5000
Name: Gender_Female, dtype: int64
1    5000
0    5000
Name: Gender_Male, dtype: int64
0    5888
1    4112
Name: bmi_class_healthy, dtype: int64
1    5590
0    4410
Name: bmi_class_overweight, dtype: int64
0    9744
1     256
Name: bmi_class_severely overweight, dtype: int64
0    9958
1      42
Name: bmi_class_underweight, dtype: int64
```

In [0]:
```
#generated data value count
for column in og_data.iloc[:, 0:6].columns:
    print(mixed_gen_df[column].value_counts())
```

```
1.0    5052
0.0    4948
Name: Gender_Female, dtype: int64
0.0    5052
1.0    4948
Name: Gender_Male, dtype: int64
0.0    5776
1.0    4224
Name: bmi_class_healthy, dtype: int64
1.0    5529
0.0    4471
Name: bmi_class_overweight, dtype: int64
0.0    9796
1.0     204
Name: bmi_class_severely overweight, dtype: int64
0.0    9965
1.0      35
Name: bmi_class_underweight, dtype: int64
```

neural network classifier

- classification based on gender

In [0]:
```
def classifier():
    model = Sequential()
    model.add(Dense(128, activation = "relu", input_dim = 3))
    model.add(Dropout(0.2))
    model.add(Dense(64, activation="relu"))
    model.add(Dropout(0.2))
    model.add(Dense(32, activation="relu"))
    model.add(Dropout(0.2))
    model.add(Dense(16, activation="relu"))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation="sigmoid"))
    model.compile(optimizer = optimizer, loss = "binary_crossentropy", metrics = ["accuracy"])
    return model

def classifier_train(classifier, x_train, y_train, epoch_limit=100, batch_size=256):

    history = classifier.fit(x_train, y_train, batch_size=batch_size, epochs=epoch_limit, verbose=0)

def evaluate(classifier, x_test, y_test):
    score = classifier.evaluate(x_test, y_test, verbose=1)
    print("Test Loss = {}, Test Accuracy = {}".format(score[0], score[1]))
```

original data

In [0]:
```
le = LabelEncoder()
gender_labels = le.fit_transform(df.Gender)
gender_labels
```

Out[0]:
```
array([1, 1, 1, ..., 0, 0, 0])
```

In [0]:
```
from sklearn.model_selection import train_test_split

X = og_data.iloc[:, 7:] #features
y = gender_labels #target labels
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size = 0.1, random_state = 42)
```

In [0]:
```
model = classifier()
classifier_train(model, x_train, y_train)
evaluate(model, x_test, y_test)
```

```
1000/1000 [==============================] - 4s 4ms/step
Test Loss = 0.22275649595260621, Test Accuracy = 0.915
```

generated data

In [0]:
```
# generated data target labels
gen_dummies = mixed_gen_df.iloc[:, :2]
s = pd.DataFrame(gen_dummies.columns[np.where(gen_dummies != 0)[1]], columns = ["Gender"])
s
```

Out[0]:

	Gender
0	Gender_Male
1	Gender_Male
2	Gender_Female
3	Gender_Male
4	Gender_Female
...	...
9995	Gender_Male
9996	Gender_Male
9997	Gender_Female
9998	Gender_Male
9999	Gender_Male

10000 rows × 1 columns

In [0]:
```
y = le.fit_transform(s)
y
```

Out[0]:
```
array([1, 1, 0, ..., 0, 1, 1])
```

In [0]:
```
X = mixed_gen_df.iloc[:, 7:] #features
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size = 0.1, random_state = 42)
```

In [0]:
```
model = classifier()
classifier_train(model, x_train, y_train)
evaluate(model, x_test, y_test)
```

```
1000/1000 [==============================] - 4s 4ms/step
Test Loss = 0.18720565822720528, Test Accuracy = 0.916
```

Results

Categorical data

- Heterogeneous data, relatively similar except for a small number (0.005% of data set) of underweight class variables that are not generated at all by the generator (same problem as previus example)
- Homogeneous data similar to original about 1% error in number of categorical variables generated

Numerical data

- Numerical data similar compared to original

Neural network classifier

- On average, the generated data is as good as, or even better, in the classification than the original data

---

Conditional GAN

- In conditional GAN you can generate data rows based on the label
- trying to solve heterogenous data problem and generating "events" based on the "patient" atributes

In [0]:
```
event17_ohe = pd.get_dummies(df["2017"])
event18_ohe = pd.get_dummies(df["2018"])
event19_ohe = pd.get_dummies(df["2019"])
event20_ohe = pd.get_dummies(df["2020"])
bmi_ohe = pd.get_dummies(df["bmi_class"])
gender_ohe = pd.get_dummies(df.Gender)

event17_ohe
```

Out[0]:

	ei jatkotoimenpiteita	laakarikaynti 1	laakarikaynti 2	laakarikaynti 3	laakarikaynti 4
0	0	0	1	0	0
1	1	0	0	0	0
2	0	0	1	0	0
3	0	1	0	0	0
4	0	0	0	0	1
...	...	...	...	...	...
9995	1	0	0	0	0
9996	0	0	0	1	0
9997	1	0	0	0	0
9998	1	0	0	0	0
9999	1	0	0	0	0

10000 rows × 5 columns

In [0]:
```
le = LabelEncoder()
labels = le.fit_transform(df.bmi_class).reshape(-1, 1)
labels
```

Out[0]:
```
array([[2],
       [0],
       [1],
       ...,
       [0],
       [0],
       [0]])
```

In [0]:
```
numerical_data = df.select_dtypes("number")
```

In [0]:
```
mms = MinMaxScaler()
numerical_data_rescaled = mms.fit_transform(numerical_data)
numerical_data_rescaled
```

Out[0]:
```
array([[0.90343165, 0.79172838, 0.863139  , 0.89533561],
       [0.82805733, 0.58695829, 0.4754764 , 0.49317406],
       [0.98706754, 0.8023644 , 0.72113127, 0.67007964],
       ...,
       [0.26720077, 0.38830089, 0.31065968, 0.38054608],
       [0.67456817, 0.59715974, 0.48298768, 0.4960182 ],
       [0.70073314, 0.31052854, 0.23843869, 0.30546075]])
```

In [0]:
```
print("event2017 ohe shape:", event17_ohe.shape)
print("event2018 ohe shape:", event18_ohe.shape)
print("event2019 ohe shape:", event19_ohe.shape)
print("event2020 ohe shape:", event20_ohe.shape)
print("gender ohe shape:", gender_ohe.shape)
print("numerical data shape:", numerical_data_rescaled.shape)
```

```
event2017 ohe shape: (10000, 5)
event2018 ohe shape: (10000, 5)
event2019 ohe shape: (10000, 5)
event2020 ohe shape: (10000, 5)
gender ohe shape: (10000, 2)
numerical data shape: (10000, 4)
```

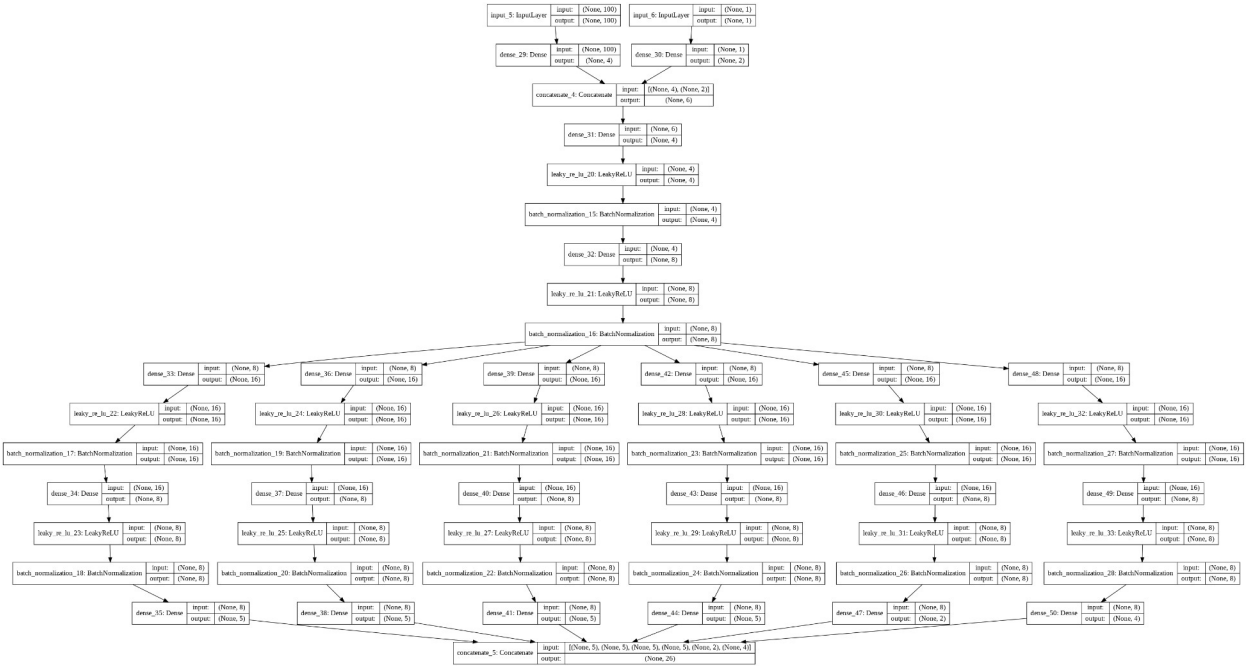
Generaattori

- Generator takes random input from the normal distribution and labels
- Generator function parameters are the size of the columns we want to generate

In [0]:
```
def build_generator(ohe1, ohe2, ohe3, ohe4, ohe5, numerical):
    #noise as input from the latent space
    noise = Input(shape = (100,))
    noise_branch = Dense(4)(noise)

    #label input
    label_input = Input(shape = (1,))
    label_branch = Dense(2)(label_input)

    #combine input branches
    merge = concatenate([noise_branch, label_branch])

    hidden_1 = Dense(4)(merge)
    hidden_1 = LeakyReLU(alpha=0.2)(hidden_1)
    hidden_1 = BatchNormalization(momentum = 0.8)(hidden_1)

    hidden_2 = Dense(8)(hidden_1)
    hidden_2 = LeakyReLU(alpha=0.2)(hidden_2)
    hidden_2 = BatchNormalization(momentum = 0.8)(hidden_2)

    #Branch 1 for generating ohe1 data
    branch_1 = Dense(16)(hidden_2)
    branch_1 = LeakyReLU(alpha=0.2)(branch_1)
    branch_1 = BatchNormalization(momentum = 0.8)(branch_1)
    branch_1 = Dense(8)(branch_1)
    branch_1 = LeakyReLU(alpha=0.2)(branch_1)
    branch_1 = BatchNormalization(momentum = 0.8)(branch_1)
    #Output 1, sofmax activation
    branch_1_output = Dense(ohe1, activation = "softmax")(branch_1)

    #Branch 2 for generating ohe2 data
    branch_2 = Dense(16)(hidden_2)
    branch_2 = LeakyReLU(alpha=0.2)(branch_2)
    branch_2 = BatchNormalization(momentum = 0.8)(branch_2)
    branch_2 = Dense(8)(branch_2)
    branch_2 = LeakyReLU(alpha=0.2)(branch_2)
    branch_2 = BatchNormalization(momentum = 0.8)(branch_2)
    #Output 2, sofmax activation
    branch_2_output = Dense(ohe2, activation = "softmax")(branch_2)

    #Branch 3 for generating ohe3
    branch_3 = Dense(16)(hidden_2)
    branch_3 = LeakyReLU(alpha=0.2)(branch_3)
    branch_3 = BatchNormalization(momentum = 0.8)(branch_3)
    branch_3 = Dense(8)(branch_3)
    branch_3 = LeakyReLU(alpha=0.2)(branch_3)
    #Output 3, softmax activation
    branch_3 = BatchNormalization(momentum = 0.8)(branch_3)
    branch_3_output = Dense(ohe3, activation = "softmax")(branch_3)

    #Branch 4 for generating ohe4
    branch_4 = Dense(16)(hidden_2)
    branch_4 = LeakyReLU(alpha=0.2)(branch_4)
    branch_4 = BatchNormalization(momentum = 0.8)(branch_4)
    branch_4 = Dense(8)(branch_4)
    branch_4 = LeakyReLU(alpha=0.2)(branch_4)
    branch_4 = BatchNormalization(momentum = 0.8)(branch_4)
    #Output 4, sofmax activation
    branch_4_output = Dense(ohe4, activation = "softmax")(branch_4)

    #Branch 5 for generating bmi ohe
    branch_5 = Dense(16)(hidden_2)
    branch_5 = LeakyReLU(alpha=0.2)(branch_5)
    branch_5 = BatchNormalization(momentum = 0.8)(branch_5)
    branch_5 = Dense(8)(branch_5)
    branch_5 = LeakyReLU(alpha=0.2)(branch_5)
    branch_5 = BatchNormalization(momentum = 0.8)(branch_5)
    #Output 4, sofmax activation
    branch_5_output = Dense(ohe5, activation = "softmax")(branch_5)

    #Branch 6 for generating numerical data
    branch_6 = Dense(16)(hidden_2)
    branch_6 = LeakyReLU(alpha=0.2)(branch_6)
    branch_6 = BatchNormalization(momentum = 0.8)(branch_6)
    branch_6 = Dense(8)(branch_6)
    branch_6 = LeakyReLU(alpha=0.2)(branch_6)
    branch_6 = BatchNormalization(momentum = 0.8)(branch_6)
    #Output 4, sofmax activation
    branch_6_output = Dense(numerical, activation = "sigmoid")(branch_6)
```

In [0]:
```
generator5 = build_generator(event17_ohe.shape[1], event18_ohe.shape[1], event19_ohe.shape[1], event20_ohe.
shape[1], gender_ohe.shape[1], numerical_data_rescaled.shape[1]) #Initialize generator
plot_model(generator5, show_layer_names = True, show_shapes = True)#Plot generator
```

Out[0]:



Discriminator

- Discriminator takes generator output (number of columns) and class label as input
- Outputs classfication (generated data or original data)

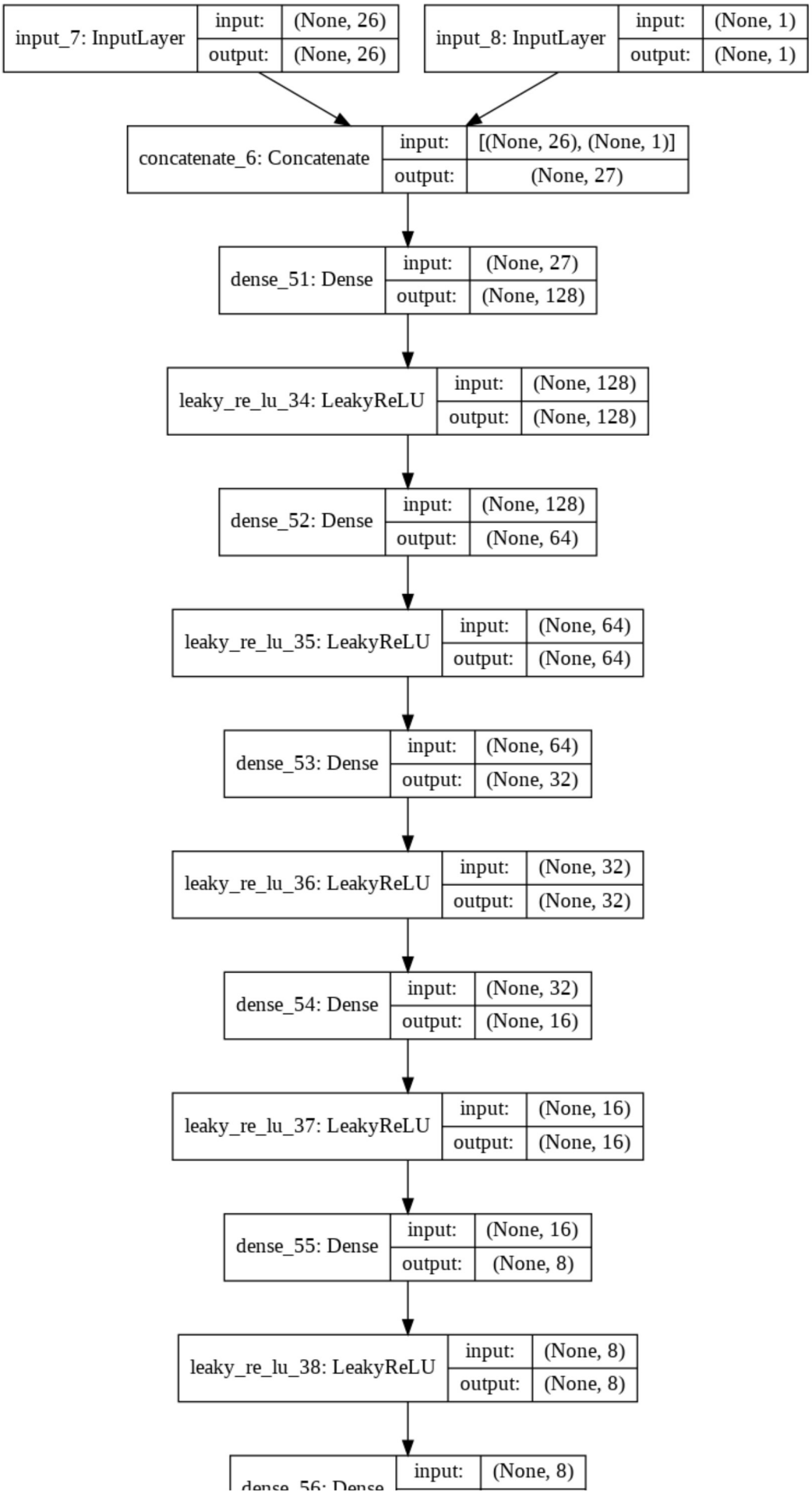
In [0]:
```
def build_discriminator(inputs_n, n_classes):
    #Inputs number of columns and labels
    data_input = Input(shape = (inputs_n,))
    label_input = Input(shape = (1,))
    #combined input
    combined = concatenate([data_input, label_input])
    d = Dense(128)(combined)
    d = LeakyReLU(alpha=0.2)(d)
    d = Dense(64)(d)
    d = LeakyReLU(alpha=0.2)(d)
    d = Dense(32)(d)
    d = LeakyReLU(alpha=0.2)(d)
    d = Dense(16)(d)
    d = LeakyReLU(alpha=0.2)(d)
    d = Dense(8)(d)
    d = LeakyReLU(alpha=0.2)(d)
    #Discriminator output for classification real or fake
    output1 = Dense(1, activation = "sigmoid")(d)
    #Compile and return model
    model = Model([data_input, label_input], output1)
    model.compile(loss = "binary_crossentropy", optimizer = optimizer, metrics = ["accuracy"])
    return model
```

In [0]:
```
optimizer = Adam(lr=0.0002, beta_1=0.5)
inputs_n = 26 #Number of columns (generator output size)
n_classes = len(np.unique(labels)) #Labels
```

In [0]:
```
discriminator5 = build_discriminator(inputs_n, n_classes) #Initialize discriminator
plot_model(discriminator5, show_layer_names = True, show_shapes = True)#Plot discriminator
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:3657:
The name tf.log is deprecated. Please use tf.math.log instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow_core/python/ops/nn_impl.py:183:
where (from tensorflow.python.ops.array_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
```

Out[0]:


input_7: InputLayer
input: (None, 26)
output: (None, 26)
input_8: InputLayer
input: (None, 1)
output: (None, 1)
concatenate_6: Concatenate
input: [(None, 26), (None, 1)]
output: (None, 27)
dense_51: Dense
input: (None, 27)
output: (None, 128)
leaky_re_lu_34: LeakyReLU
input: (None, 128)
output: (None, 128)
dense_52: Dense
input: (None, 128)
output: (None, 64)
leaky_re_lu_35: LeakyReLU
input: (None, 64)
output: (None, 64)
dense_53: Dense
input: (None, 64)
output: (None, 32)
leaky_re_lu_36: LeakyReLU
input: (None, 32)
output: (None, 32)
dense_54: Dense
input: (None, 32)
output: (None, 16)
leaky_re_lu_37: LeakyReLU
input: (None, 16)
output: (None, 16)
dense_55: Dense
input: (None, 16)
output: (None, 8)
leaky_re_lu_38: LeakyReLU
input: (None, 8)
output: (None, 8)
input: (None, 8)

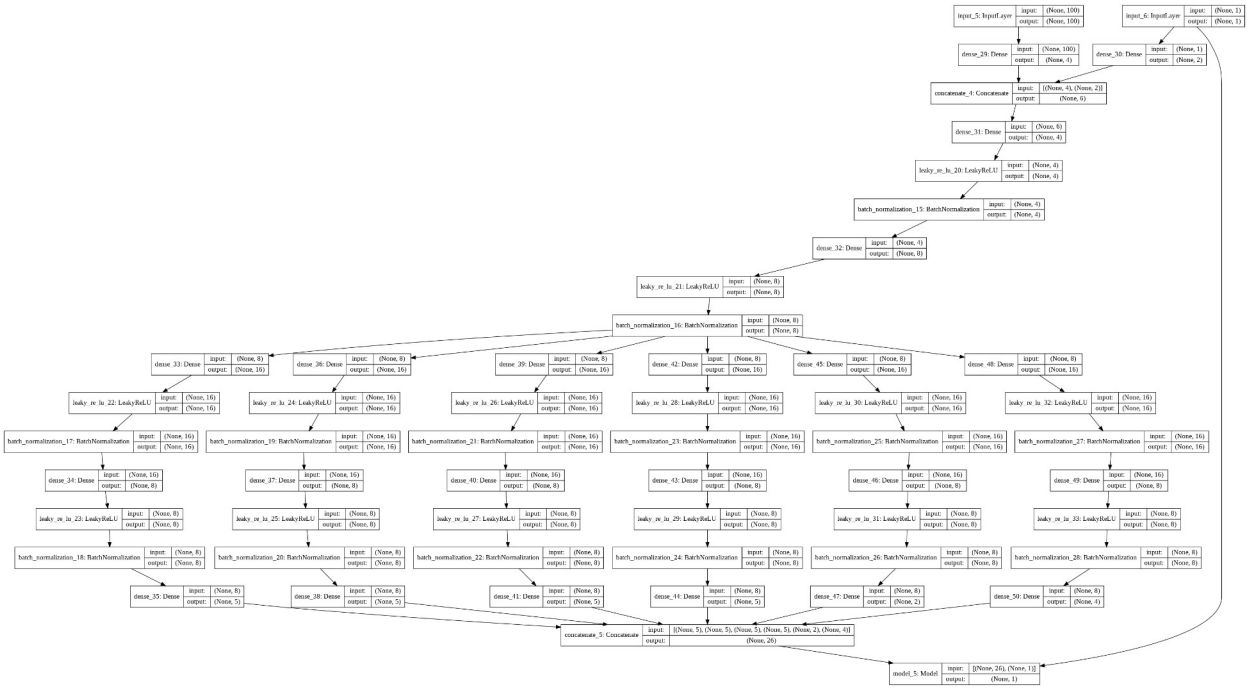
GAN

- input is generator input
- output is dicriminator output

In [0]:
```
def build_gan(generator, discriminator):
    #Make discriminator not trainable
    discriminator.trainable = False
    #Get generator input
    generator_noise, generator_label = generator.input
    #Get generator output
    generator_output = generator.output
    #Gan output (discriminator output)
    gan_output = discriminator([generator_output, generator_label])
    #Initialize gan
    model = Model([generator_noise, generator_label], gan_output)
    #Compile model
    model.compile(loss = "binary_crossentropy", optimizer = optimizer)
    #Return Model
    return model
```

In [0]:
```
gan5 = build_gan(generator5, discriminator5) #Initialize GAN
plot_model(gan5, show_layer_names = True, show_shapes = True) #Plot GAN
```

Out[0]:



CGAN training

In [0]:
```
def train(gan, generator, discriminator, ohe1, ohe2, ohe3, ohe4, gender_ohe, numerical_data, labels, latent
_dim, n_epochs, n_batch, n_eval):
    #get half batch size to update discriminator
    half_batch = int(n_batch / 2)
    #lists for stats from the model training
    discriminator_loss = []
    generator_loss = []
    #generate class labels for fake and real
    valid = np.ones((half_batch, 1))
    fake = np.zeros((half_batch, 1))
    y_gan = np.ones((n_batch, 1))
    #training loop
    for i in range(n_epochs):

        #select random batch from the real data
        idx = np.random.randint(0, numerical_data.shape[0], half_batch)
        ohe1 = ohe1[idx]
        ohe2 = ohe2[idx]
        ohe3 = ohe3[idx]
        ohe4 = ohe4[idx]
        gender_ohe = gender_ohe[idx]
        numerical_data = numerical_data[idx]
        real_labels = labels[idx]

        #concatenate data for discriminator input (same way as the generator output)
        real_data = np.concatenate([ohe1, ohe2, ohe3, ohe4, gender_ohe, numerical_data], axis = 1)

        #generate fake samples from the noise and fake labels
        noise = np.random.normal(0, 1, (half_batch, latent_dim))
        fake_labels = np.random.randint(0, 4, (half_batch, 1))
        fake_data = generator.predict([noise, fake_labels])

        #train the discriminator and return losses
        d_loss_real, _ = discriminator.train_on_batch([real_data, real_labels], valid)
        d_loss_fake, _ = discriminator.train_on_batch([fake_data, fake_labels], fake)

        #append stats from training
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
        discriminator_loss.append(d_loss)

        #train generator
        noise = np.random.normal(0, 1, (n_batch, latent_dim))
        noise_labels = np.random.randint(0, 4, (n_batch, 1))
        g_loss= gan.train_on_batch([noise, noise_labels], y_gan)
        generator_loss.append(g_loss)

        #evaluate progress
        if (i+1) % n_eval == 0:
            print ("Epoch: %d [Discriminator loss: %f] [Generator loss: %f]" % (i + 1, d_loss, g_loss))

    #plot losses
    plt.figure(figsize = (20, 10))
    plt.plot(discriminator_loss, label = "Discriminator loss")
    plt.plot(generator_loss, label = "Generator loss")
    plt.title("Stats from training GAN")
    plt.grid()
    plt.legend()
```

In [0]:
```
train(gan5, generator5, discriminator5, event17_ohe.values, event18_ohe.values, event19_ohe.values, event20
_ohe.values, gender_ohe.values, numerical_data_rescaled, labels, latent_dim = 100, n_epochs = 20000, n_batc
h = 2575, n_eval = 100)
```

```
Epoch: 100 [Discriminator loss: 0.030941] [Generator loss: 3.399959]
Epoch: 200 [Discriminator loss: 0.104493] [Generator loss: 2.835105]
Epoch: 300 [Discriminator loss: 0.030399] [Generator loss: 3.643715]
Epoch: 400 [Discriminator loss: 0.013073] [Generator loss: 4.369587]
Epoch: 500 [Discriminator loss: 0.006619] [Generator loss: 4.933706]
Epoch: 600 [Discriminator loss: 0.004745] [Generator loss: 5.703031]
Epoch: 700 [Discriminator loss: 0.002376] [Generator loss: 6.234666]
Epoch: 800 [Discriminator loss: 0.001290] [Generator loss: 6.790228]
Epoch: 900 [Discriminator loss: 0.000873] [Generator loss: 7.195805]
Epoch: 1000 [Discriminator loss: 0.000679] [Generator loss: 7.516388]
Epoch: 1100 [Discriminator loss: 0.000860] [Generator loss: 7.462957]
Epoch: 1200 [Discriminator loss: 0.000676] [Generator loss: 7.689208]
Epoch: 1300 [Discriminator loss: 0.000573] [Generator loss: 7.822417]
Epoch: 1400 [Discriminator loss: 0.000471] [Generator loss: 8.082445]
Epoch: 1500 [Discriminator loss: 0.000397] [Generator loss: 8.292772]
Epoch: 1600 [Discriminator loss: 0.000464] [Generator loss: 8.306713]
Epoch: 1700 [Discriminator loss: 0.000318] [Generator loss: 8.572988]
Epoch: 1800 [Discriminator loss: 0.000235] [Generator loss: 8.811868]
Epoch: 1900 [Discriminator loss: 0.000194] [Generator loss: 8.979785]
Epoch: 2000 [Discriminator loss: 0.000163] [Generator loss: 9.139801]
Epoch: 2100 [Discriminator loss: 0.000141] [Generator loss: 9.302763]
Epoch: 2200 [Discriminator loss: 0.000114] [Generator loss: 9.449041]
Epoch: 2300 [Discriminator loss: 0.000098] [Generator loss: 9.597330]
Epoch: 2400 [Discriminator loss: 0.000087] [Generator loss: 9.658030]
Epoch: 2500 [Discriminator loss: 0.000072] [Generator loss: 9.835332]
Epoch: 2600 [Discriminator loss: 0.000064] [Generator loss: 9.965568]
Epoch: 2700 [Discriminator loss: 0.000057] [Generator loss: 10.099340]
Epoch: 2800 [Discriminator loss: 0.000051] [Generator loss: 10.313618]
Epoch: 2900 [Discriminator loss: 0.000044] [Generator loss: 10.399305]
Epoch: 3000 [Discriminator loss: 0.000037] [Generator loss: 10.551877]
Epoch: 3100 [Discriminator loss: 0.000034] [Generator loss: 10.688606]
Epoch: 3200 [Discriminator loss: 0.000028] [Generator loss: 10.747960]
Epoch: 3300 [Discriminator loss: 0.000026] [Generator loss: 10.862705]
Epoch: 3400 [Discriminator loss: 0.000023] [Generator loss: 11.048045]
Epoch: 3500 [Discriminator loss: 0.000020] [Generator loss: 11.189514]
Epoch: 3600 [Discriminator loss: 0.000017] [Generator loss: 11.319454]
Epoch: 3700 [Discriminator loss: 0.000016] [Generator loss: 11.394788]
Epoch: 3800 [Discriminator loss: 0.000014] [Generator loss: 11.447070]
Epoch: 3900 [Discriminator loss: 0.000012] [Generator loss: 11.618692]
Epoch: 4000 [Discriminator loss: 0.000011] [Generator loss: 11.728607]
Epoch: 4100 [Discriminator loss: 0.000011] [Generator loss: 11.785288]
Epoch: 4200 [Discriminator loss: 0.000010] [Generator loss: 11.999336]
Epoch: 4300 [Discriminator loss: 0.000008] [Generator loss: 12.116876]
Epoch: 4400 [Discriminator loss: 0.000007] [Generator loss: 12.206695]
Epoch: 4500 [Discriminator loss: 0.000006] [Generator loss: 12.307136]
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-80-e0a6fbb55383> in <module>()
----> 1 train(gan5, generator5, discriminator5, event17_ohe.values, event18_ohe.values, event19_ohe.value
s, event20_ohe.values, gender_ohe.values, numerical_data_rescaled, labels, latent_dim = 100, n_epochs = 2
0000, n_batch = 2575, n_eval = 100)

<ipython-input-50-66bcf70858dd> in train(gan, generator, discriminator, ohe1, ohe2, ohe3, ohe4, gender_oh
e, numerical_data, labels, latent_dim, n_epochs, n_batch, n_eval)
     41         noise = np.random.normal(0, 1, (n_batch, latent_dim))
     42         noise_labels = np.random.randint(0, 4, (n_batch, 1))
---> 43         g_loss= gan.train_on_batch([noise, noise_labels], y_gan)
     44         generator_loss.append(g_loss)
     45

/usr/local/lib/python3.6/dist-packages/keras/engine/training.py in train_on_batch(self, x, y, sample_weig
ht, class_weight)
   1447             ins = x + y + sample_weights
   1448         self._make_train_function()
-> 1449         outputs = self.train_function(ins)
   1450         return unpack_singleton(outputs)
   1451

/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py in __call__(self, inputs)
   2977                     return self._legacy_call(inputs)
   2978
-> 2979             return self._call(inputs)
   2980         else:
   2981             if py_any(is_tensor(x) for x in inputs):

/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py in _call(self, inputs)
   2935             fetched = self._callable_fn(*array_vals, run_metadata=self.run_metadata)
   2936         else:
-> 2937             fetched = self._callable_fn(*array_vals)
   2938         return fetched[:len(self.outputs)]
   2939

/usr/local/lib/python3.6/dist-packages/tensorflow_core/python/client/session.py in __call__(self, *args,
**kwargs)
   1470         ret = tf_session.TF_SessionRunCallable(self._session._session,
   1471                                                self._handle, args,
-> 1472                                                run_metadata_ptr)
   1473         if run_metadata:
   1474           proto_data = tf_session.TF_GetBuffer(run_metadata_ptr)

KeyboardInterrupt:
```

In [0]:
```
def generate_data(class_num, num_samples):
    noise = np.random.normal(0, 1, (num_samples, 100))
    labels = np.asarray([class_num for _ in range(num_samples)]).reshape(-1, 1)
    generated_data = generator5.predict([noise ,labels])
    return generated_data
```

In [0]:
```
le.classes_
```

In [0]:
```
df.bmi_class.value_counts() #count label variables from the original data
```

In [0]:
```
ov = generate_data(0, 5590)
he = generate_data(1, 4112)
so = generate_data(2, 256)
uv = generate_data(3, 42)
```

In [0]:
```
zero = np.random.randint(0,1, (len(ov), 1))
one = np.random.randint(1,2, (len(he), 1))
two = np.random.randint(2,3, (len(so), 1))
three = np.random.randint(3,4, (len(uv), 1))
```

In [0]:
```
ov = np.append(ov, zero, axis = 1)
he = np.append(he, one, axis = 1)
so = np.append(so, two, axis = 1)
uv = np.append(uv, three, axis = 1)
```

In [0]:
```
# original data
og_df = pd.concat([event17_ohe, event18_ohe, event19_ohe, event20_ohe, gender_ohe, numerical_data, df.bmi_c
lass], axis = 1)
og_df
```

In [0]:
```
#generated data
generate_data = np.concatenate([ov, he, so, uv])
generate_data.shape
```

In [0]:
```
#generated data to dataframe
gen_df = pd.DataFrame(data = generate_data, columns = og_df.columns)
gen_df
```

In [0]:
```
#round data
gen_df.iloc[:, :22] = np.round(gen_df.iloc[:, :22])
gen_df.iloc[:, 22:26] = mms.inverse_transform(gen_df.iloc[:, 22:26])
gen_df
```

Comaparing original and generated data

- Normal distribution
- mean, std, var
- correlation matrix

In [0]:
```
for column in gen_df.iloc[:, 22:26]:
    print(column, "Normal distribution")
    normal_distribution(og_df[column], gen_df[column])
```

In [0]:
```
#correlation matrix compareing
print("Original data")
plt.figure(figsize = (30, 20))
sns.heatmap(og_df.corr(), annot = True)
```

In [0]:
```
print("Generated data")
plt.figure(figsize = (30, 20))
sns.heatmap(gen_df.corr(), annot = True)
```